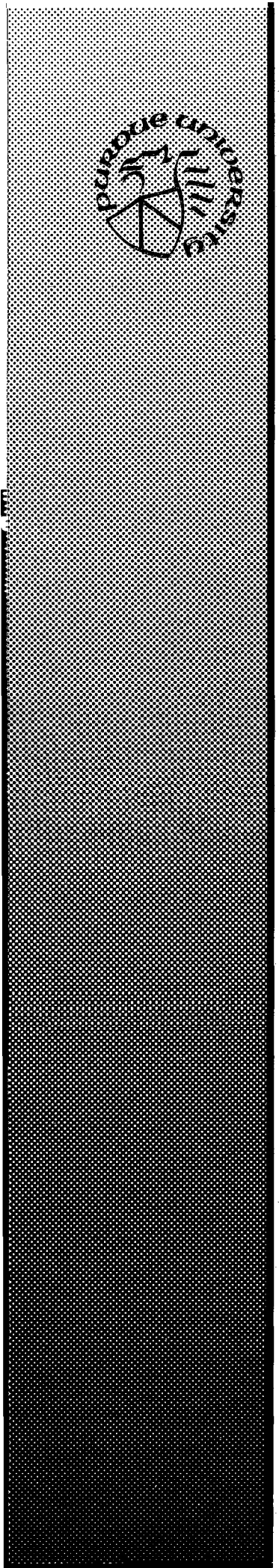
11-1-1987

Residue Arithmetic VLSI Array Architecture for Manipulator Pseudo-Inverse Jacobian Computation

P. R. Chang
Purdue University

C. S. G. Lee
Purdue University



Residue Arithmetic VLSI Array Architecture for Manipulator Pseudo-Inverse Jacobian Computation

P. R. Chang
C. S. G. Lee

TR-EE 87-42
November 1987

School of Electrical Engineering
Purdue University
West Lafayette, Indiana 47907

Residue Arithmetic VLSI Array Architecture for Manipulator Pseudo-Inverse Jacobian Computation

P. R. Chang and C. S. G. Lee

Ransburg Robotics Laboratory
School of Electrical Engineering
Purdue University
West Lafayette, Indiana 47907

TR-EE 87-42

November 1987

ABSTRACT

Most Cartesian-based control strategies require the computation of the manipulator inverse Jacobian in real time at every sampling period. In some cases, the Jacobian matrix is not of full column or row rank due to singularity or redundant robot configuration. This requires the computation of the manipulator pseudo-inverse Jacobian in real time. The calculation of the pseudo-inverse Jacobian may become extremely sensitive to small perturbation in the data and numerical instabilities, when the Jacobian matrix is not of full column or row rank. Even if the Jacobian matrix is of full rank, the ill-conditioned problem may still plague the computation of the pseudo-inverse Jacobian. This paper presents the use of residue arithmetic for the *exact* computation of the manipulator pseudo-inverse Jacobian to obviate the roundoff errors normally associated with the computations. A two-level macro-pipelined residue arithmetic array architecture implementing the Decell's pseudo-inverse algorithm has been developed to overcome the ill-conditioned problem of the pseudo-inverse computation. Furthermore, the Decell algorithm is quite suitable for VLSI array implementation to achieve the real-time computation requirement. The first-level arrays are data-driven, wavefront-like arrays and perform the matrix multiplications, matrix diagonal additions, and trace computations. A pool or sequence of the first-level arrays are then configured into a second-level macro-pipeline with outputs of one array acting as inputs to another array in the pipe. The proposed architecture can calculate the pseudo-inverse Jacobian with a pipelined time in the same computational complexity order as evaluating a matrix product in a wavefront array.

This work was supported in part by the NSF-Purdue Engineering Research Center under Grant CDR8500022. Any opinions, findings, and conclusions or recommendations expressed in this article are those of the authors and do not necessarily reflect the views of the funding agency.

1. Introduction

Robot manipulators are highly nonlinear systems and their control involves actuating appropriate joint motors due to small changes in the *pose* (position and orientation) of the manipulator hand along a planned Cartesian trajectory. This requires the computation of the manipulator inverse Jacobian at every sampling period in real time. The manipulator Jacobian is a linear mapping which relates the derivatives of the joint-variables to the derivatives of the manipulator hand coordinates. The pose of the manipulator hand relative to a fixed inertial coordinate system can be described by a set of 6 algebraic equations containing the joint variables $q_1, q_2, \cdots, q_n$ as

$$\mathbf{x} = \mathbf{f}(\mathbf{q}) \tag{1}$$

where $\mathbf{x} \triangleq (p_x, p_y, p_z, \phi_x, \phi_y, \phi_z)^T$ is a 6-dimensional vector describing the pose of the manipulator hand with respect to the inertial coordinate frame, $\mathbf{q}$ is the n-dimensional joint-variable vector for an n-link manipulator, and the superscript "T" indicates matrix/vector transpose. Differentiating Eq. (1) with respect to time yields

$$\dot{\mathbf{x}}(t) \triangleq \begin{bmatrix} \mathrm{v}(t) \\ \omega(t) \end{bmatrix} = \mathbf{J}(\mathbf{q})\dot{\mathbf{q}}(t) \tag{2}$$

where $\mathrm{v}(t)$ and $\omega(t)$ are, respectively, the linear and angular velocities of the manipulator hand, $\dot{\mathbf{q}} \triangleq (\dot{q}_1, \dot{q}_2, \cdots, \dot{q}_n)^T$ is the joint-velocity vector of the manipulator, and $\mathbf{J}(\mathbf{q})$ is the $6\times n$ Jacobian matrix (usually $n \geq 6$) and is a function of joint variables.

In general, the manipulator Jacobian (or forward Jacobian) is quite easy to determine than the inverse Jacobian. However, most Cartesian-based control techniques require the computation of the inverse Jacobian in real time at every control servo period. In particular, the use of the inverse Jacobian in resolved motion rate control [1], and the model-based control techniques in Cartesian space [2,3], has been a computational bottleneck for these control schemes. In order to use these control schemes effectively, efficient computation of the inverse Jacobian in real time must be developed.

Most of the existing methods in computing the inverse Jacobian are formulated to be computed by uniprocessor computers [4-6]. One of the most widely used methods in computing the inverse Jacobian is first by determining the forward Jacobian and then taking its inverse. Several efficient parallel algorithms have been developed for computing the forward Jacobian [7-9]. If the manipulator has six degrees-of-freedom and is nonsingular at $\mathbf{q}(t)$, then the Jacobian matrix may be inverted using normal matrix inversion techniques for square matrices. For example, in the case of resolved motion rate control, the joint-velocity vector can be computed from the manipulator hand velocities as

$$\dot{\mathbf{q}}(t) = \mathbf{J}^{-1}(\mathbf{q})\dot{\mathbf{x}}(t) \tag{3}$$

where $\mathbf{J}^{-1}(\mathbf{q})$ is the 6×6 inverse Jacobian matrix. If the manipulator Jacobian is singular at $\mathbf{q}(t)$, then the inverse of the Jacobian is not defined because the Jacobian matrix

is not of full column or row rank. In the case of redundant robots, the inverse of the Jacobian is also not defined because the Jacobian matrix is rectangular. Useful solutions to the above cases, however, can be obtained through the use of pseudo-inverse [10-13]. The best approximate solution of the inverse Jacobian to Eq. (3) is given by the pseudo-inverse [12,13] and is denoted by $\mathbf{J}^{+}(\mathbf{q})$. Using the pseudo-inverse, Eq. (3) can be written as [12,13]

$$\dot{\mathbf{q}}(t) = \mathbf{J}^{+}(\mathbf{q})\,\dot{\mathbf{x}}(t) + (\mathbf{I} - \mathbf{J}^{+}(\mathbf{q})\,\mathbf{J}(\mathbf{q}))\,\mathbf{z} \tag{4}$$

where $\mathbf{I}$ is an $n \times n$ identity matrix and $\mathbf{z}$ is an arbitrary vector in the joint-velocity space. Equation (4) indicates that the resultant joint velocities can be decomposed into a combination of the least-squares solution of minimum norm plus a homogeneous solution created by the action of a projection operator $(\mathbf{I} - \mathbf{J}^{+}\,\mathbf{J})$†, which describes the redundancy of the manipulator system. If the Jacobian is a square matrix and non-singular at $\mathbf{q}(t)$, then the projection operator is equal to the null operator and Eq. (4) reduces to Eq. (3).

Many numerical algorithms, such as the Decell algorithm, the Grevill algorithm, the Hermite algorithm, and the Gram-Schmidt orthogonalization algorithm, have been developed to compute the pseudo-inverse $\mathbf{J}^{+}$ [12,13]. However, the calculations involved in these four algorithms are quite sensitive to ill-conditioning in which the *exact* solution is extremely sensitive to small perturbations in the data. Some researchers [14-16] considered the possibility of using the residue arithmetic for the exact computation of pseudo-inverses in order to obviate the roundoff errors normally associated with their computations. These four algorithms have been implemented in the residue number system (RNS) for this purpose. Unfortunately, [15] showed that the Grevill algorithm is not suitable for implementation in the residue arithmetic system. This is because the algorithm requires the greatest common divisor and the least common multiple in each iteration, otherwise the moduli choice may change very much. The Hermite algorithm and the Gram-Schmidt orthogonalization algorithm require a lot of data broadcasting operations which are not feasible for implementation in VLSI array architectures. The Decell algorithm [14,15], based on the Cayley-Hamilton theorem, is an iterative procedure with a sequence of matrix operations and can compute the pseudo-inverse $\mathbf{J}^{+}$ quite efficiently. This sequence of matrix operations can be easily implemented in the residue arithmetic and is quite suitable for VLSI implementation [17]. This paper presents the design of a two-level macro-pipelined VLSI array architecture for the real-time computation of the *exact* solution of the pseudo-inverse Jacobian using the Decell algorithm in the residue arithmetic system.

The first-level array of the proposed architecture is an asynchronous data-driven, wavefront-like array [17] and performs the matrix multiplication, matrix diagonal addition, and the trace computation required in the Decell algorithm. The topological

† For brevity, we shall drop the function dependency of $\mathbf{q}$ on all Jacobian matrices and their inverse.

configuration of the first-level array is a two-dimensional processor array consisting of three basic components: (1) an ordinary matrix multiplication wavefront array, (2) an adder tree, and (3) I/O memory modules. The adder tree, connected to the diagonal processing elements (PEs) of the ordinary wavefront array, is used to evaluate the trace computation, and the calculation of the matrix multiplication and the trace computation can be executed in parallel. The above executions are in an asynchronous data-driven manner, and the data transfers between a PE and its immediate neighbor are by mutual convenience or handshaking protocol. A pool or sequence of the first-level arrays are then configured into the second-level macro-pipeline with the outputs of one array acting as inputs to another array in the pipe. For computing the pseudo-inverse of a general $p \times n$ $(n \geq p)$ matrix $\mathbf{J}$, the second-level macro-pipelined linear array is a pipe of p stages and can evaluate a total of p iterations of the Decell algorithm with the first-level array in each stage. The second-level linear array is a synchronous systolic array whose data movements in the array are controlled by global timing-reference clock signal. The results of each stage are stored in its local memory and waiting for activated global synchronous signals. The pipelined time of the proposed two-level pipelined array architecture has a computational order of $O(n + 2p - 1)$ which is the same computational complexity order as evaluating a matrix product in an ordinary wavefront array. Furthermore, the primitive processing elements of the architecture perform computations in the residue number system. Recently, [18] proposed attractive wafer-scale integration (WSI) techniques for implementing the residue number system designs. WSI technology is based on the concept of placing one or more functional processing units on a semiconductor wafer. The processing units are then interconnected on the wafer as opposed to individual packaging. The wafer-scale integration compresses a large microelectronics representing a complete digital system onto a single wafer, and its primary advantages are an improvement in total system density and a built-in fault-tolerance capability. Thus, the WSI has the potential to provide a very cost-effective and reliable way of implementing the proposed RNS two-level macro-pipelined array architecture for the computation of manipulator pseudo-inverse Jacobian in real time.

2. Unpleasant Fact for Calculating the Pseudo-Inverse

If the manipulator Jacobian is not of full column or row rank, an unpleasant fact exists for calculating the pseudo-inverse Jacobian $\mathbf{J}^+$. The rank of $\mathbf{J}$ which cannot be determined exactly because it can be easily altered by an arbitrarily small perturbation. This problem is commonly called the ill-conditioned problem for which the *exact* solution is extremely sensitive to small perturbations in the data. In solving such problems, the introduction of roundoff errors in the computations can be disastrous. This ill-conditioned problem can be explained in the following lemma and the proof of the lemma can be found in [13].

Unpleasant Fact Lemma [13]. Suppose a matrix $\mathbf{J} \in R^{p \times n}$ is of neither full column nor row rank, then for any real number k and any $\epsilon > 0$, there exists a matrix $\mathbf{E}$,

$||\mathbf{E}|| < \epsilon$, such that $||(\mathbf{J}+\mathbf{E})^{+} - \mathbf{J}^{+}|| \geq k$.

Even if the Jacobian matrix is of full rank, the ill-conditioned problem may still plague the computation of the pseudo-inverse Jacobian. For example, consider the problem of evaluating the determinant of the matrix

$$\mathbf{J} = \begin{bmatrix} -73 & 78 & 24 \\ 92 & 66 & 25 \\ -80 & 37 & 10 \end{bmatrix}.$$

This problem is extremely ill-conditioned as shown by the fact that if the roundoff error matrix is

$$\mathbf{E} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 10^{-2} \end{bmatrix}$$

then we have $\det(\mathbf{J}) = 1$, whereas $\det(\mathbf{J}+\mathbf{E}) = -118.94$. In other words, if an accumulation of roundoff errors in the computations were to correspond to the introduction of the perturbation matrix $\mathbf{E}$, then the computed value of the determinant would be -118.94, whereas the exact value is 1, even though the matrix $\mathbf{J}$ is of full rank.

The above lemma and example show that roundoff-error-free computations are necessary for evaluating the pseudo-inverse Jacobian matrix. It is known that the residue number system (RNS) is the most popular technique for error-free computations in signal processing and filter design [19]. We shall apply the residue arithmetic computation technique for computing the "error-free" pseudo-inverse Jacobian.

3. Overview of Residue Number System

The residue number system (RNS) arithmetic is a familiar concept in the number theory. In this section, we shall review some definitions and properties of the residue number system that will lead us to the computation of the error-free pseudo-inverse Jacobian.

Definition 1. Given any integer x and any modulus m, if $r \equiv x \pmod{m}$ and $0 \leq r < m$, then we write $r = |x|_m$ and say r is a residue of x modulo m.

With this definition, computations in single-modulus residue arithmetic is simple in the sense that no negative integers are involved in the RNS. However, if we wish to solve problems involving negative integers, we must be able to handle negative integers as well as positive integers. One way to accomplish this is to introduce the system of symmetric residues modulo m.

Definition 2. Given any integer x and any modulus m, if $r \equiv x \pmod{m}$ and $-\frac{1}{2}m \leq r \leq \frac{1}{2}m$, then we write $r = /x/_m$ and say r is a symmetric residue of x

modulo m.

It can be shown that both residue representations of x are unique. Furthermore, there exists a conversion between $|x|_m$ and $/x/_m$,

$$|x|_m = \begin{cases} /x/_m \ , & \text{if } 0 \leq /x/_m < \dfrac{m}{2} \\ /x/_m + m \ , & \text{otherwise} \end{cases} \tag{5}$$

and

$$/x/_m = \begin{cases} |x|_m \ , & \text{if } 0 \leq |x|_m < \dfrac{m}{2} \\ |x|_m - m \ , & \text{otherwise} \end{cases} . \tag{6}$$

Some important properties of the RNS are listed below:

(i) $|x \pm y|_m = | \, |x|_m \pm |y|_m \, |_m = |x \pm |y|_m \, |_m$.

(ii) $|x \, y|_m = | \, |x|_m \, |y|_m \, |_m$.

(iii) $|k \, x|_m = |ky|_m$ and $\gcd(k,m) = 1$, then $|x|_m = |y|_m$, where $\gcd(k,m) \triangleq$ greatest common divisor of k and m.

Next, we introduce the concept of multiplicative inverse modulo m.

Definition 3. Assume x, y, and $m > 1$ are integers, and if $0 < y < m$ and $|xy|_m = |yx|_m = 1$, then we write $y = x^{-1}(m)$ and say y is a multiplicative inverse of x modulo m.

Notice that regardless of the sign of x, the multiplicative inverse of x modulo m is defined to be a positive integer in the range of $0 < x^{-1}(m) < m$. Of course, there are questions of existence and uniqueness of the multiplicative inverse modulo m, and these questions are addressed by the following theorem.

Theorem 1 [20]. If x and $m > 1$ are integers, then $x^{-1}(m)$ exists and is unique, if and only if $|x|_m \neq 0$ and $\gcd(x,m) = 1$.

The choice of the modulo m is quite important. If m is not a prime number, then no convenient explicit expression for $x^{-1}(m)$ has been discovered, whereas if m is a prime number, one can find a convenient explicit expression for it.

Theorem 2 [Fermat] [20]. If x and $m > 1$ are integers, and if m is a prime number and $x^{-1}(m)$ exists, then $x^{-1}(m) = |x^{m-2}|_m = |(|x|_m)^{m-2}|_m$.

This theorem indicates that we can perform division to a limited extent. For example, if the quotient of two integers is again an integer, we can compute this quotient as follows: If x, y, and $m > 1$ are integers, and if $x^{-1}(m)$ exists, then $|\frac{y}{x}|_m = |yx^{-1}(m)|_m$, even in those cases when x does not divide y. For example,

$\left|\frac{7}{9}\right|_{11} = \left|7 \cdot 9^{-1}(11)\right|_{11} = \left|7 \cdot 5\right|_{11} = 2$. Finally, it should be mentioned that the above properties are also suitable for the symmetric residue representation.

For any integer x, there are two different fixed-radix weighted number representations: the signed number representation and the unsigned number representation. For the signed number system, both positive and negative number numbers can appear. Hence all integer values are assumed to be within the following range depending on whether m is odd or even

$$x \in \left[-\frac{m}{2}, \quad \frac{m}{2} - 1\right] \quad , \quad \text{for } m \text{ is even}$$

$$x \in \left[-\frac{(m-1)}{2}, \quad \frac{(m-1)}{2}\right] \quad , \quad \text{for } m \text{ is odd} .$$

However, it is more convenient to use what appears to be only positive numbers by mapping the negative range above the positive upper limit. This can be accomplished by the following rule: If x is a negative integer that falls within the above negative range, it can be represented by the positive number as,

$$m - |x| = m + x, \quad x < 0$$

which exceeds the positive upper limit but falls below m. Thus, any integer value greater than the upper positive range limit represents a negative number. For example, assume $m = 60$ and $x_{signed} = -2$, then $x_{unsigned} = 58$. It should be noted that these two representations are identical, usually the signed number system is used in the symmetric residue system and the unsigned number system is used in the common residue system.

3.1. Arithmetic Decomposition

Arithmetic representations can be decomposed linearly into an equivalent one containing many components, each with a relatively small arithmetic range [18,20]. This decomposition leads us to a number of parallel and independent pseudo-inverse computations, each using the residue arithmetic which allows us to use short registers and simple arithmetic operations. Furthermore, since the decomposition is linear, the final pseudo-inverse results for each parallel computation can be linearly recombined back together, yielding the desired pseudo-inverse result. To achieve this objective, we need to introduce the concept of multiple-modulus residue arithmetic.

Definition 4. Let $m_1, m_2, \cdots, m_L$ be the bases for a residue number system, where $\gcd(m_i, m_j) = 1$, for $i \neq j$, and let $M = m_1 m_2 \cdots m_L = \prod_{i=1}^{L} m_i$. The unique L-tuple residue arithmetic representation of x is given by

$$x \sim \{|x|_{m_1}, |x|_{m_2}, \cdots, |x|_{m_L}\} \quad \text{or} \quad x \sim \{r_1, r_2, \cdots, r_L\} .$$

where $r_i = |x|_{m_i}$. Notice that the representation also holds true for the symmetric residue system. Similar to the RNS, there are some important properties for the multiple-modulus residue arithmetic:

(i) Assume $x \sim \{|x|_{m_1}, \cdots, |x|_{m_L}\}$ and $y \sim \{|y|_{m_1}, \cdots, |y|_{m_L}\}$
then $|x \pm y|_M \sim \{|x \pm y|_{m_1}, \cdots, |x \pm y|_{m_L}\}$
and $|x\,y|_M \sim \{|x\,y|_{m_1}, |x\,y|_{m_2}, \cdots, |x\,y|_{m_L}\}$.

(ii) Multiplicative inverse

(a) $x^{-1}(M) \sim \{x^{-1}(m_1), x^{-1}(m_2), \cdots, x^{-1}(m_L)\}$.

(b) If $m_1, m_2, \cdots, m_L$ are all prime numbers, then $x^{-1}(M) \sim \{|x^{m_1-2}|_{m_1}, |x^{m_2-2}|_{m_2}, \ldots, |x^{m_L-2}|_{m_L}\}$.

(iii) Quotient of two integers

(a) $|\frac{x}{y}|_M \sim \{|xy^{-1}(m_1)|_{m_1}, |xy^{-1}(m_2)|_{m_2}, \ldots, |xy^{-1}(m_L)|_{m_L}\}$.

(b) If, in addition, the bases are all prime numbers, then
$|\frac{x}{y}|_M \sim \{|xy^{m_1-2}|_{m_1}, |xy^{m_2-2}|_{m_2}, \ldots, |xy^{m_L-2}|_{m_L}\}$.

As an example for illustrating the idea of the multiple-modulus residue arithmetic, let $M = 60 = 3\times4\times5$ and $x_{unsigned} = 58$, then, $m_1 = 3$, $m_2 = 4$, $m_3 = 5$ and $|x|_{m_1} = 1$, $|x|_{m_2} = 2$, $|x|_{m_3} = 3$. Thus, we have $x \sim \{1, 2, 3\}$.

3.2. Arithmetic Recomposition (or Reconstruction)

Assume that the L-tuple residue representation of x, $x \sim \{|x|_{m_1}, |x|_{m_2}, \cdots, |x|_{m_L}\}$, is given, one wants to find the corresponding value of x. Two techniques are available to achieve this purpose: the Chinese remainder theorem (CRT) and the mixed-radix recomposition (MRR).

3.2.1. The Chinese Remainder Theorem

Let $m_1, m_2, \cdots, m_L$ be the bases for a residue system where $\gcd(m_i, m_j) = 1$, for $i \neq j$, and $M = \prod_{i=1}^{L} m_i$. Also, let $\hat{m}_j = M/m_j$, $1 \leq j \leq L$. If x has the residue representation $x \sim \{r_1, \cdots, r_i, \cdots, r_L\}$, where $r_i = |x|_{m_i}$, $1 \leq i \leq L$, then $|x|_M = |\sum_{j=1}^{L} \hat{m}_j |r_j\, \hat{m}_j^{-1}(m_j)|_{m_j}|_M$. Furthermore, x is expressed in the unsigned number system and falls within the range $[0, M-1]$. Thus, $x = |x|_M$. Although the CRT method is fairly simple in principle [20], it would not be able in its given form to be implemented in a residue computer, since the residue computer is equipped to perform arithmetic operations modulo m_i, not modulo M operation as required by the CRT. In contrast, the mixed-radix recomposition procedure can be implemented in a residue

machine because it requires modulo m_i operations only.

3.2.2. The Mixed-Radix Recomposition

Szabo and Tanaka [20] proposed the mixed-radix recomposition (MRR) method to avoid the modulo M addition operation in the CRT. The MRR is of great importance in the residue arithmetic computation and the VLSI implementation for the following two reasons:

(i) The mixed-radix system is a weighted number system and magnitude comparison can be easily performed. Also, it can be shown that its VLSI implementation is quite simple.

(ii) Conversion from the residue to a certain mixed-radix system is relatively fast in residue computers.

The mixed-radix number system can be described as follows: Let $m_1, m_2, \cdots, m_L$ be the pairwise relatively prime bases (or radices) for the mixed-radix system. A number x may be expressed in the mixed-radix form as

$$x = a_L \prod_{i=1}^{L-1} m_i + \cdots + a_3 m_1 m_2 + a_2 m_1 + a_1 \tag{7}$$

where the a_i are the mixed-radix digits and $0 \leq a_i < m_i$. For a given set of radices, the mixed-radix representation of x is denoted by $<a_L, a_{L-1}, \cdots, a_1>$ where the digits are listed in order of decreasing significance. It can be shown that $x \in [0, \prod_{i=1}^{L} m_i - 1] = [0, M-1]$ and has a unique representation. Notice that if x is in the signed number system, the digits a_i should be within the interval $(-\frac{1}{2} m_i, \frac{1}{2} m_i)$, and then $x \in (-\frac{1}{2}M, \frac{1}{2}M)$. For convenience, we will only discuss the case for the unsigned number system.

Assume that the L-tuple residue representation of x, $\{r_1, r_2, \cdots, r_L\}$, is given, one wants to find the corresponding mixed-radix digits a_i, in terms of r_i, $1 \leq i \leq L$. The MRR method involves the following steps:

(i) To obtain a_1, Eq. (7) is first taken as modulo m_1. Since all terms except the last are multiples of m_1, we have,

$$a_1 = |x|_{m_1} = r_1 . \tag{8}$$

(ii) To obtain a_2, we first form $x - a_1$ in its residue code, then take modulo m_2 on both sides of the following equation,

$$(x - a_1) \bmod m_2 = \{(a_L \prod_{i=3}^{L-1} m_i + \cdots + a_3)\, m_2 m_1 + m_1 a_2\} \bmod m_2$$

$$= m_1 a_2 \bmod m_2 . \tag{9.a}$$

Because $\gcd(m_1, m_2) = 1$ and from Theorem 1, $m_1^{-1}(m_2)$ exists, so we have

$$a_2 \bmod m_2 = \mid m_1^{-1}(m_2)(\mid x \mid_{m_2} - a_1) \mid_{m_2} = \mid m_1^{-1}(m_2)(r_2 - r_1) \mid_{m_2}. \quad (9.b)$$

Furthermore, since $0 \leq a_2 < m_2$, we have

$$a_2 = a_2 \bmod m_2 = \mid C_{12}(r_2 - r_1) \mid_{m_2} \quad (9.c)$$

where $C_{12} = m_1^{-1}(m_2)$.

(iii) For any j, and $1 \leq j \leq L$, r_j is given and $a_1, a_2, \cdots, a_{j-1}$ have been evaluated already. Thus, by induction, we have

$$\begin{aligned} a_j &= \mid (\cdots(((r_j - a_1)m_1^{-1}(m_j) - a_2)\, m_2^{-1}(m_j))\cdots) - a_{j-1})m_{j-1}^{-1}(m_j) \mid_{m_j} \\ &= \mid (\cdots(((r_j - a_1)C_{1j} - a_2)C_{2j})\cdots) - a_{j-1})\, C_{j-1,j} \mid_{m_j} \end{aligned} \quad (10)$$

where $C_{ij} = m_i^{-1}(m_j)$.

It should be noted that if we are interested in the signed number system, then the procedure is the same as the above equations except the symmetric residue $/x/_m$ replaces the $\mid x \mid_m$.

The above residue arithmetic for integers can be extended to integral matrices. We shall define the residue matrix modulo m and most of the results that we have discussed for integers can be extended to integral matrices.

Definition 5. Let $\mathbf{X} = [x_{ij}]$ be a $p \times n$ integral matrix and $m > 1$ be an integer. If $\mathbf{R} = [r_{ij}]$ is the matrix with elements defined by $r_{ij} = \mid x_{ij} \mid_m$ for all i and j, then we write $\mathbf{R} = \mid \mathbf{X} \mid_m$ and say $\mathbf{R}$ is a residue of $\mathbf{X}$ modulo m.

We shall use this residue matrix modulo m concept in the residue number system to compute the error-free pseudo-inverse Jacobian.

4. Computation of Pseudo-Inverse Jacobian Using RNS

The Decell algorithm for residue computation consists of a sequence of matrix multiplications, matrix diagonal additions, and trace computations. These operations are suitable for VLSI array implementation [17]. Stallings and Boullion [14] proposed using the residue arithmetic to compute the Decell algorithm. They used the Chinese remainder theorem (CRT) to recover the final solution. However, using the CRT for recomposition involves the modulo M addition, which has the potential of resulting in long registers and complicated arithmetic hardware. A better solution is to use the mixed-radix recomposition (MRR) technique to avoid the modulo M addition by using normal binary adders.

We shall first describe the concept of the Decell algorithm and then propose a VLSI array implementation for it. Throughout this paper, we assume that $\mathbf{A}$ is a $p \times n$ matrix and $n \geq p$, otherwise we could compute $(\mathbf{A}^T)^+$ using the relation $\mathbf{A}^+ = [(\mathbf{A}^T)^+]^T$. The Decell algorithm for computing the pseudo-inverse $\mathbf{A}^+$ is based on the Cayley-Hamilton

theorem and consists of computing a sequence of matrices $\mathbf{A}_0, \mathbf{A}_1, \cdots, \mathbf{A}_K$, as follow:

$$\begin{aligned}
&\mathbf{A}_0 = \mathbf{0}, && q_0 = -1, && \mathbf{B}_0 = \mathbf{I}, \\
&\mathbf{A}_1 = \mathbf{A}\,\mathbf{A}^T, && q_1 = tr(\mathbf{A}_1), && \mathbf{B}_1 = \mathbf{A}_1 - q_1\mathbf{I}, \\
&\mathbf{A}_2 = \mathbf{A}_1\,\mathbf{B}_1, && q_2 = \frac{1}{2}\,\mathrm{tr}(\mathbf{A}_2), && \mathbf{B}_2 = \mathbf{A}_2 - q_2\mathbf{I}, \\
&\quad\vdots && \quad\vdots && \quad\vdots \\
&\mathbf{A}_{K-1} = \mathbf{A}_1\,\mathbf{B}_{K-2}, && q_{K-1} = \frac{1}{K-1}\,\mathrm{tr}(\mathbf{A}_{K-1}), && \mathbf{B}_{K-1} = \mathbf{A}_{K-1} - q_{K-1}\mathbf{I}, \\
&\mathbf{A}_K = \mathbf{A}_1\,\mathbf{B}_{K-1}, && q_K = \frac{1}{K}\,\mathrm{tr}(\mathbf{A}_K), && \mathbf{B}_K = \mathbf{A}_K - q_K\mathbf{I},
\end{aligned} \tag{11}$$

The pseudo-inverse $\mathbf{A}^+$ is given by

$$\mathbf{A}^+ = \frac{1}{q_K}\,\mathbf{A}^T\mathbf{B}_{K-1} \tag{12}$$

where $\mathrm{tr}(\mathbf{A}_i) \triangleq$ trace of $\mathbf{A}_i$, $1 \le i \le K$, $\mathbf{I}$ is a $p \times p$ identity matrix, and K is the rank of $\mathbf{A}$. Since the rank of $\mathbf{A}$ is not known apriori, the iteration is continued until the matrix product $\mathbf{A}_1\mathbf{B}_i$ becomes a zero matrix for some i, $i = 1, 2, \cdots, K$. The termination of the iteration will determine K as well as the rank of $\mathbf{A}$. The critical step in the Decell algorithm involves the determination of whether or not $\mathbf{A}_1\mathbf{B}_i = \mathbf{0}$ for some i, $i = 1, 2, \cdots, K$. Obviously, the error-free computation is useful in overcoming the numerical instability in this iteration as well as the determination of the zero matrix in the matrix product. Thus, we shall use the residue arithmetic for the computation of the Decell algorithm in an effort to obviate the round-off errors that may cause difficulty in determining the zero matrix from the matrix product $\mathbf{A}_1\mathbf{B}_i$.

The use of residue arithmetic presumes that each element a_{ij} of the matrix $\mathbf{A}$ is an integer. Although fixed word-length computers store only rational numbers, they can be converted to integers by an appropriate scaling. For example, in the radix u system, a_{ij} can be evaluated by $a_{ij} = \sum_{k=-s}^{s} \alpha_{ij}^{(k)} u^i = u^{-s}\left(\sum_{k=0}^{2s} \alpha_{ij}^{(k-s)} u^k\right) = u^{-s}\,\bar{a}_{ij}$, where $\bar{a}_{ij} = \sum_{k=0}^{2s} \alpha_{ij}^{(k-s)} u^k$ is the normalized integer and u^{-s} is the constant scaling factor. Thus, we can obtain the normalized matrix $\bar{\mathbf{A}} = u^s\,\mathbf{A}$. Since it is known that $\mathbf{A}^+ = (u^{-s}\cdot\bar{\mathbf{A}})^+ = \frac{1}{u^{-s}}\,\bar{\mathbf{A}}^+ = u^s\,\bar{\mathbf{A}}^+$, one may apply the proposed residue arithmetic Decell algorithm to obtain the pseudo-inverse $\bar{\mathbf{A}}^+$ from which the desired pseudo-inverse $\mathbf{A}^+$ may be obtained by multiplying the scaling factor u^s. For convenience, we assume that the elements of $\mathbf{A}$ are integers in the following discussion.

Re-examining the equations (11) and (12) carefully again, one may find that the determination of whether or not $\mathbf{A}_1\mathbf{B}_i = \mathbf{0}$ for some i is a very critical step. If the ordinary fixed-point arithmetic is used, it will be extremely difficult, in the presence of

roundoff errors, to determine whether or not the elements of each iteration in $\mathbf{A}_1\mathbf{B}_i$ are exactly zero. Hence, the algorithm may suffer from the numerical instability. So, the error-free computation must be used in Eq. (11) for overcoming this difficulty. However, the evaluation of Eq. (12) can be done in the ordinary fixed-point arithmetic. Thus, the residue arithmetic Decell algorithm can be divided into two parts: Eq. (11) is evaluated in the residue arithmetic and Eq. (12) is evaluated in the ordinary fixed-point arithmetic.

Before discussing the residue arithmetic Decell algorithm, one has to consider the proper choice of the range M and the selection of the bases (or moduli) m_i, $1 \leq i \leq L$, which are critical to the success of the method for calculating $\mathbf{A}^+$. Stallings and Boullion [14] suggested the following criterion for selecting M

$$M > \max \{X^p \ , p(p-K+1)X^{p-1}\} \tag{13}$$

where $X = \min \{\mathrm{tr}(\mathbf{A}\mathbf{A}^T) \ , \ ||\mathbf{A}\mathbf{A}^T||\}$ and $K =$ rank $(\mathbf{A})$. For simplicity, two practical criteria are given as follows:

(i) $$M \geq 2 \prod_{i=1}^{p} \left(\sum_{j=1}^{n} a_{ij}^2\right)^{1/2} \tag{14}$$

where a_{ij} is the $(i \ , j)$ element of $\mathbf{A}$.

(ii) $$M > p^{p^2/2} X^{p^2} \tag{15}$$

where $X =$ the maximal absolute value of an element in the matrix $\mathbf{A}\mathbf{A}^T$. Stallings and Boullion [14] also suggested that the bases (or moduli) m_i, $1 \leq i \leq L$, should be chosen to be large prime numbers greater than p and must satisfy $M = \prod_{i=1}^{L} m_i$, then rank $(|\mathbf{A}|_{m_i}) =$ rank $(|\mathbf{A}\mathbf{A}^T|_{m_i}) =$ rank $(\mathbf{A}\mathbf{A}^T) =$ rank $(\mathbf{A}) = K$, where $1 \leq i \leq L$. In other words, the rank of $|\mathbf{A}|_{m_i}$ is equal to rank $(\mathbf{A})$ regardless of the modulus choice m_i. This indicates that the residue arithmetic Decell iteration will terminate at the same $(K+1)$th iteration for any different m_i, where K is the rank of $\mathbf{A}$. Thus, we can use the multiple-modulus arithmetic to calculate the pseudo-inverse. However, the last $(p+1)$th iteration is not necessary when the rank of $\mathbf{A}$ is equal to p. Since $K \leq \min(p \ , n) = p$, if the matrix multiplications $\mathbf{A}_j = \mathbf{A}_1\mathbf{B}_{j-1}$ are not equal to zero matrices for the previous p iterations, where $1 \leq j \leq p$, then $\mathbf{A}_{p+1}$ of the last $(p+1)$th iteration must be a zero matrix, and K is determined automatically and is equal to p, otherwise K would be greater than p which is impossible. Figure 1 indicates that the Decell algorithm may be decomposed into L parallel Decell algorithms, each of which can be implemented using arithmetic modulo the respective component of M. The recombination of the outputs from the L parallel systems may be achieved by using the mixed-radix recomposition technique. The computational procedure of the residue arithmetic Decell algorithm can be described in the following Residue Arithmetic Decell Algorithm:

Algorithm RADA (*Residue Arithmetic Decell Algorithm*). Given a $p \times n$ matrix $\mathbf{A}$ with normalized integral elements, $p \leq n$, the range M, and the moduli m_i , $1 \leq i \leq L$, this algorithm uses the residue arithmetic to compute the pseudo-inverse $\mathbf{A}^+$ based on the Decell algorithm as in Eqs. (11) and (12).

R1 [Initialization.] Let $\mathbf{A}_0^i = \mathbf{0}$, $q_0^i = -1$, $\mathbf{B}_0^i = \mathbf{I}$, $1 \leq i \leq L$.

R2 [Arithmetic Decomposition.]
FOR $i = 1$ STEP 1 UNTIL L DO

(2.a) Obtain $|\mathbf{A}|_{m_i}$,

(2.b) Perform $\mathbf{A}_1^i = ||\mathbf{A}|_{m_i} |\mathbf{A}^T|_{m_i}|_{m_i}$.

R3 [Decell's Iteration.]
FOR $j = 1$ STEP 1 UNTIL p DO

R4 Perform $\mathbf{A}_j^i = |\mathbf{A}_1^i \mathbf{B}_{j-1}^i|_{m_i}$.

R5 [Termination.]

(i) If $\mathbf{A}_j^i = \mathbf{0}$, then stop and let $K_i = j-1$ and go to Step R2. Otherwise, continue.

(ii) If $j = p$, then stop and let $K_i = p$ and go to Step R2. Otherwise, continue.

R6 Perform $q_j^i = ||j^{-1}|_{m_i} |\mathrm{tr}(\mathbf{A}_j^i)|_{m_i}|_{m_i}$
(notice that $|j^{-1}|_{m_i}$ are pre-computed integer values)

R7 Perform $\mathbf{B}_j^i = |\mathbf{A}_j^i - q_j^i \mathbf{I}|_{m_i}$
End Do_Block of Step R3
End Do_Block of Step R2

$K = K_1 = \cdots = K_i \cdots = K_L$.

R8 [Arithmetic Recomposition.] Reconstruct q_K and $\mathbf{B}_{K-1}$ from q_K^i and $\mathbf{B}_{K-1}^i$, $1 \leq i \leq L$, by using the MRR technique (notice that K is the rank of $\mathbf{A}$)

R9 [Compute the pseudo-inverse $\mathbf{A}^+$.] Compute $\mathbf{A}^+ = \dfrac{1}{q_K} \mathbf{A}^T \mathbf{B}_{K-1}$ in the ordinary fixed-point system.

R10 [Output and terminate.] Output the pseudo-inverse $\mathbf{A}^+$ and terminate.

END RADA.

Let us give an example to illustrate the residue arithmetic Decell algorithm. Assume

$$\mathbf{A} = \begin{bmatrix} 1 & 0 & 50 & 0 \\ 0 & 2 & 0 & 1 \\ 1 & 0 & 0 & 0 \end{bmatrix}$$

and choose $m_1 = 7$, $m_2 = 11$, $m_3 = 17$, and $m_4 = 23$, thus

$M = 7 \times 11 \times 17 \times 23 = 30107$ satisfies the criteria of Eq. (14) or Eq. (15). The following discussion is in the signed number system. First, we compute $/q_K^1/_7$ and $/\mathbf{B}_{K-1}^1/_7$

$$\mathbf{A}_1^1 = //\mathbf{A}/_7/\mathbf{A}^T/_7/_7 = \begin{bmatrix} 2 & 0 & 1 \\ 0 & -2 & 0 \\ 1 & 0 & 1 \end{bmatrix}$$

$$q_1^1 = /\mathrm{tr}(\mathbf{A}_1^1)/_7 = 1\ , \mathbf{B}_1^1 = /\mathbf{A}_1^1 - q_1^1\mathbf{I}/_7 = \begin{bmatrix} 1 & 0 & 1 \\ 0 & -3 & 0 \\ 1 & 0 & 0 \end{bmatrix}$$

$$\mathbf{A}_2^1 = /\mathbf{A}_1^1\mathbf{B}_1^1/_7 = \begin{bmatrix} 3 & 0 & 2 \\ 0 & -1 & 0 \\ 2 & 0 & 1 \end{bmatrix}$$

$$q_2^1 = /\frac{1}{2}\mathrm{tr}(\mathbf{A}_2^1)/_7 = //2^{-1}/_7/\mathrm{tr}(\mathbf{A}_2^1)/_7/_7 = /(-3)\cdot(3)/_7 = -2$$

$$\mathbf{B}_2^1 = /\mathbf{A}_2^1 - q_2^1\mathbf{I}/_7 = \begin{bmatrix} -2 & 0 & 2 \\ 0 & 1 & 0 \\ 2 & 0 & 3 \end{bmatrix}$$

$$\mathbf{A}_3^1 = /\mathbf{A}_1^1\mathbf{B}_2^1/_7 = \begin{bmatrix} -2 & 0 & 0 \\ 0 & -2 & 0 \\ 0 & 0 & -2 \end{bmatrix}$$

$$q_3^1 = /\frac{1}{3}\mathrm{tr}(\mathbf{A}_3^1)/_7 = /(5)\cdot(-6)/_7 = -2\ ,$$

$\mathbf{A}_4^1 = \mathbf{A}_1^1\mathbf{B}_3^1 = \mathbf{0}$. Terminate the RADA algorithm and obtain the resultant $/q_3/_7$ and $/\mathbf{B}_2/_7$. Similarly, repeat the above computations with moduli 11, 17, and 23, we obtain, respectively, $/q_3/_{11} = 4$, $/q_3/_{17} = 5$, $/q_3/_{23} = 11$, and

$$/\mathbf{B}_2/_{11} = \begin{bmatrix} 5 & 0 & -5 \\ 0 & 3 & 0 \\ -5 & 0 & -2 \end{bmatrix}\ , /\mathbf{B}_2/_{17} = \begin{bmatrix} 5 & 0 & -5 \\ 0 & 1 & 0 \\ -5 & 0 & -7 \end{bmatrix}\ ,\ /\mathbf{B}_2/_{23} = \begin{bmatrix} 5 & 0 & -5 \\ 0 & -7 & 0 \\ -5 & 0 & -7 \end{bmatrix}$$

So, the multiple-modulus residue representations are

$$q_3 \sim \{-2, 4, 5, 11\}\ ,\ \mathbf{B}_2 \sim \begin{bmatrix} \{-2,5,5,5\} & \{0,0,0,0\} & \{2,-5,-5,-5\} \\ \{0,0,0,0\} & \{1,3,1,-7\} & \{0,0,0,0\} \\ \{2,-5,-5,-5\} & \{0,0,0,0\} & \{3,-2,-7,-7\} \end{bmatrix}$$

One may use the MRR technique to reconstruct the q_3 and $\mathbf{B}_2$ with the precomputed weighting factors, i.e., $C_{12} = -3$, $C_{13} = 5$, $C_{14} = 10$, $C_{23} = -3$, $C_{24} = -2$, and $C_{34} = -4$, where $C_{ij} = m_i^{-1}(m_j)$. For example, the residue representation of q_3 is given, find the associated mixed-radix digits $<a_4,a_3,a_2,a_1>$ where the mixed-radix expression is $q_3 = a_4(7\times11\times17) + a_3(7\times11) + a_2\times(7) + a_1$. The associated mixed-radix digits may be found by using Eqs. (8)-(10) in the signed number system, then we have

$a_1 = -2$, $a_2 = 4$, $a_3 = -8$ and $a_4 = 10$. Furthermore, q_3 can be evaluated as $q_3 = 12500$. Similarly, using the same procedures, the elements of $\mathbf{B}_2$ can be evaluated as $b_{12} = b_{21} = b_{23} = b_{22} = 0$, $b_{11} = 5$, $b_{13} = -5$, $b_{31} = -5$, $b_{22} = 2500$, $b_{33} = 12505$, where b_{ij} is the $(i\,,\,j)$ element of $\mathbf{B}_2$. Finally, we can compute $\mathbf{A}^+$ in the ordinary fixed-point system as follows:

$$\mathbf{A}^+ = \frac{1}{q_3}\mathbf{A}^T\mathbf{B}_2 = \frac{1}{12500}\begin{bmatrix} 1 & 0 & 1 \\ 0 & 2 & 0 \\ 50 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix}\begin{bmatrix} 5 & 0 & -5 \\ 0 & 2500 & 0 \\ -5 & 0 & 12505 \end{bmatrix} = \begin{bmatrix} 0. & 0. & 1. \\ 0. & 0.4 & 0. \\ 0.02 & 0. & -0.02 \\ 0. & 0.2 & 0. \end{bmatrix}$$

5. Hardware Implementation of Residue Arithmetic

The implementation of the multiple-modulus residue arithmetic requires the consideration of three hardware components: (i) basic residue arithmetic processing cells, (ii) arithmetic decomposition cell, and (iii) arithmetic recomposition cell. It should be noted that the implementations have been previously realized by VLSI/WSI techniques [18].

5.1. Basic Residue Arithmetic Processing Cells

Two basic residue arithmetic processing cells are residue adder and multiplier. As shown in Figure 2, one may use two k-bit binary adders to realize a modulo m residue adder, where $2^k < m$. Assuming that x and y are inputs to the first binary adder of the residue adder, and if the result of the first binary adder overflows, it can be corrected for the modulo m operation by adding $(-m)$ to the result of the first addition. If the first addition does not overflow, it may fall in the range $[\,m\,,\,2^k-1\,]$, in which case it can also be corrected by adding $(-m)$ to the first addition. The carry bits generated from the first binary adder or the second binary adder indicate whether or not $(x+y)$ is greater than m. A multiplexor controlled by the carry selects the correct output.

One familiar method of residue multiplication is "log transform-residue addition - antilog transform" [18]. The concept of logarithms can be used for the multiplication of field elements. The theory guarantees that there is at least one primitive element which generates all nonzero field elements [20]. It is known that when m is a prime number [20], the nonzero elements of $GF(m)$ will be $GF^*(m) = \{1,2,\cdots,m-1\}$. For each $i \in GF^*(m)$, there is a corresponding element e, where $0 \le e < m-1$, such that $i = \mu^e$, where μ is a primitive element of $GF(m)$. The exponent is the logarithm of the element

$$i = \mu^e \quad \text{and} \quad \log i = e\ . \tag{16}$$

The multiplication of two field elements is equivalent to the addition modulo $(m-1)$ of the corresponding exponents. If $i_1 = \mu^{e_1}$ and $i_2 = \mu^{e_2}$, then

$$|i_1 \cdot i_2|_m = \mu^{|e_1+e_2|_{(m-1)}} \quad \text{and} \quad |e_1 + e_2|_{(m-1)} = \log(i_1 \cdot i_2) \tag{17}$$

Thus, multiplications can be implemented by adding the appropriate exponents as determined from a logarithm table [21]. The procedure requires three steps: (1) find the exponents e_i from the logarithm table, (2) add indices and take modulo $(m-1)$, and (3) find antilogarithm in the logarithm table to yield the result. This procedure is shown in Figure 3.

5.2. Arithmetic Decomposition Cell

The decomposition of an input number consists of finding its residues for its various moduli m_i. In general, this does not create problems because the input number can be applied simultaneously to separate decomposition blocks as suggested in Figure 4.a. Each decomposition block can be realized by a custom VLSI design based on a sequential restoring division algorithm [18,22]. This yields a compact design that is easily modified to simplify the 2's complement to the RNS conversion process. The division algorithm simplifies considerably when only the remainder is of interest. The resulting architecture can be a pipeline requiring only subtractions. Assuming that the input is a k-bit positive number, we can subtract from it the binary representation of the modulus m_i such that the highest order "1" bit of the representation is aligned with the highest order input bit. If the result is negative, the original input is passed on (i.e. subtract 0). If the result is positive, the subtracted version is passed on. In either case, a $(k-1)$-bit number will result. This number is now treated as the input, m_i is shifted one less position to the left, and the above process repeats until m_i has been totally shifted. The remainder is the result after that step. A floor plan of a decomposition block to find the modulo 5 residue is shown in Figure 4.b. With 2's complement input values, a positive input is passed directly to the residue decomposers, and 0 is added to their outputs. For a negative input, the input is first complemented (without adding one) and then passed to the residue decomposer.

5.3. Arithmetic Recomposition Cell

Recomposition using the MRR process is shown in Figure 5 for a four-modulus system. The need for the modulo M addition required for the CRT can be eliminated. Pipelined operation is possible so that a new value may enter every clock cycle. In the recomposition architecture, the values C_{ij} and w_i are pre-computed and stored in a read-only memory (ROM), where $w_k = \prod_{i=0}^{k-1} m_i$ and $m_1 = 1$. Once, the residue inputs r_i, $1 \leq i \leq 4$, enter into the architecture, the residue adders and multiplier are executed from top to down. The buffers are used to synchronize the timing of the pipelines. Thus, the resulting mixed-radix digits $< a_4, a_3, a_2, a_1 >$ are carried out at the same time. Later on, a tree-structured architecture with normal binary adders and multipliers can evaluate the resulting binary output from the mixed-radix digits.

Let us examine how the MRR process of Eqs. (8)-(10) can be realized by the proposed pipelined implementation. From Eq. (8), the resulting a_1 is equal to the input r_1, so r_1 would be shifted down through the buffers. Observing Eqs. (9.c) and (10), the resulting r_1 must also be sent to the right three pipes for evaluating a_2, a_3, and a_4. These pipes are executed in their corresponding modulus residue number systems. For example, the resulting a_2 may be carried out and present at the second stage of the pipe after a modulo m_2 residue subtraction, $x = |r_2 - r_1|_{m_2}$, and a modulo m_2 residue multiplication, $a_2 = |C_{12}x|_{m_2}$. At the same time, the pipes for a_3 and a_4 are also executed in the similar operations of modulo m_3 and m_4, respectively. These operations correspond to the computations within the most inside parenthesis of Eqs. (9.c) and (10). Next, the resulting a_2 will be stored in the buffers of its pipe and also broadcasted to the other two pipes. This process will continue to make the computations within the parentheses of Eq. (10) for a_3 and a_4 to be executed from inside to outside. Because of the multiple-pipe structure, the calculations of a_2, a_3, and a_4 would be evaluated in parallel. Examining the MRR architecture, one notes that multiplications are only required by the moduli m_i, $i = 2, \cdots, L$ (in this case, $L = 4$). Thus, the best implementation is to assign m_1 the largest valued modulus and m_L the smallest valued modulus. This allows for short register (or buffers) and simple arithmetic operations with relatively small arithmetic ranges, which can be implemented on the proposed pipelined architecture.

6. VLSI Architecture for Residue Arithmetic Decell Algorithm

From the first part of the Decell algorithm (i.e. Eq. (11)), one finds that there are three different matrix operations involved in the iteration of the algorithm: matrix multiplication, trace computation, and matrix diagonal addition. Based on the discussion in section 4, it is suggested that these operations in Eq. (11) should be done in the residue number system so that the exact determination of $\mathbf{A}_1\mathbf{B}_i = \mathbf{0}$ for some i can be easily detected. The determination of $\mathbf{A}_1\mathbf{B}_i = \mathbf{0}$ terminates further iterations, and a matrix multiplication and a scalar division in Eq. (12) are then executed in the ordinary fixed-point arithmetic.

Using the multiple-modulus arithmetic to implement the Decell algorithm requires the algorithm be decomposed into L identical parallel algorithms, each of which is implemented using arithmetic modulo m_i and $\prod_{i=1}^{L} m_i = M$. The results from the L parallel algorithms can be reconstructed by the MRR technique to obtain the desired result. In this section, we shall discuss one of the L identical parallel algorithms.

There are at most $(K+1)$ iterations involved in Eq. (11). However, K is not known until $\mathbf{A}_1\mathbf{B}_i$ becomes a zero matrix for some i. Thus, K or the rank of $\mathbf{A}$ will be determined at the end of the iterations. In fact, the rank of $\mathbf{A}$ is always less than or equal to $\min(p,n)$, that is, $K \leq \min(p,n)$. If $p \leq n$, then $K \leq p$, and we have at most $(p+1)$ iterations. However, the algorithm RADA is section 4 indicates that the last $(p+1)$th

iteration is not necessary. Based on this fact, it is possible to implement Eq. (11) on a linear array of p processing architectures as shown in Figure 6. Each processing architecture is an asynchronous data-driven, wavefront-like, two-dimensional array. Each performs the matrix multiplication, matrix diagonal addition, and trace computation involved in one iteration of Eq. (11). Furthermore, the architecture also has to determine whether or not $\mathbf{A}_1 \mathbf{B}_K$ is a zero matrix which is used to determine K and terminate the iterations as well. Once K is known, no more operations will be executed. In this case, inhibited tags are assigned to those output data q_{K+1}, $\mathbf{B}_{K+1}$, and $\mathbf{B}_K$ of the $(K+1)$th processing architecture with $q_{K+1} = q_K$, $\mathbf{B}_K = \mathbf{B}_{K-1}$, and they are also used to disable the computations of the next $(p-K-1)$ architectures, where q_k and $\mathbf{B}_{K-1}$ are the input data. When a processing architecture is disable, no operations will be executed, and the input data are stored in its local memory and it is waiting for the activated global synchronous signal. The above design of Figure 6 is regarded as a two-level macro-pipelined array: the first-level consists of data-driven, wavefront-like, two-dimensional arrays (or the processing architectures), and the second-level consists of cascading p of these arrays into a linear macro-pipelined array. In the second-level pipelined array, the outputs of one array are acting as inputs to another array in the pipe. Furthermore, it should be noted that the first-level array is in an asynchronous data-driven manner, whereas the second-level linear array is a synchronized multiprocessing in which the data movements in the array are controlled by global timing-reference clock signal.

After $(p-K-1)$ clock periods, the desired output data q_K, $\mathbf{B}_{K+1}$, and $\mathbf{B}_{K-1}$ from the $(K+1)$th first-level array will be pumped out of the last pth first-level array of the second-level pipe. Then, we use the q_K and $\mathbf{B}_{K-1}$ to calculate the pseudo-inverse $\mathbf{A}^+ = \frac{1}{q_K}\mathbf{A}^T \mathbf{B}_{K-1}$. Since the proposed architecture is a two-level macro-pipeline, the desired results are pumped out in succession and at every clock period coming in. Next, we shall discuss our design of the first-level, asynchronous data-driven, wavefront array.

Kung [17] proposed an asynchronous, data-driven, wavefront array to implement matrix operations, in particular the matrix multiplication. The wavefront array combines the systolic pipelining principle and the data-flow computing concept. In fact, it should be noted that the major difference between a wavefront array and a systolic array is the data-driven property. There is no global timing reference in a wavefront array, and yet the order of task sequencing is correctly followed. In the wavefront architecture, the data transfer between a processing element (PE) and its immediate neighbors is by mutual convenience. Whenever data are available, the transmitting PE informs the receivers, and the receivers accept the data whenever required. They then communicate with the sender to acknowledge that the data have been received. This scheme can be implemented by means of a simple handshaking protocol [17] which ensures that the computational wavefronts propagate in an orderly manner instead of crashing into one another. Since there is no need to synchronize the entire array, a

wavefront array is architecturally scalable. Kung [17] demonstrated how a square matrix multiplication algorithm can be executed on a square, orthogonal wavefront array. This concept can be extended to a more general case to perform the multiplication of non-square matrices. If $\mathbf{A}$ is an $m \times n$ matrix and $\mathbf{B}$ is an $n \times p$ matrix, the wavefront array size needed is $m \times p$, thus the computations are completed at time $n + (m-1) + (p-1) = n + m + p - 2$. Extending this idea to our non-square matrix multiplications for $\mathbf{A}_1 = \mathbf{A}\mathbf{A}^T$ of Eq. (11) and $\mathbf{A}^T \mathbf{B}_{K-1}$ of Eq. (12), where $\mathbf{A}$ is a $p \times n$ matrix and $\mathbf{B}_{K-1}$ is a $p \times p$ matrix, and $n \geq p$, one may use the wavefront arrays of size $p \times p$ and size $n \times p$ to evaluate the $\mathbf{A}\mathbf{A}^T$ and $\mathbf{A}^T \mathbf{B}_{K-1}$, respectively, with the same computational time of $(n + 2p - 2)$.

With some modifications of the wavefront array, it is possible to evaluate the matrix multiplication, trace computation, and matrix diagonal addition for one iteration of Eq. (11) by the proposed array in the residue number system as shown in Figure 7. Furthermore, the proposed array also includes the determination of whether or not $\mathbf{A}_1 \mathbf{B}_i$ is a zero matrix. To explain the functions of the proposed array clearly, we shall first discuss the main-frame of the processor array which does not include the adder tree and dotted connected lines. This main-frame of the processor array is mainly used for matrix multiplications only. It is quite important to illustrate how the matrix multiplication is executed on a square, orthogonal $p \times p$ wavefront array. Let $\mathbf{A} = [a_{ij}]$, $\mathbf{B} = [b_{ij}]$, and $\mathbf{C} = \mathbf{A}\mathbf{B} = [C_{ij}]$ be all $p \times p$ matrices. The matrix $\mathbf{A}$ can be decomposed into columns $\mathbf{A}_i$ and the matrix $\mathbf{B}$ into rows $\mathbf{B}_j$, and we have

$$\mathbf{C} = \mathbf{A}_1 \mathbf{B}_1 + \mathbf{A}_2 \mathbf{B}_2 + \cdots + \mathbf{A}_p \mathbf{B}_p \tag{18}$$

where the product $\mathbf{A}_i \mathbf{B}_i$ is the "outer product." The matrix multiplication can then be carried out in p sets of wavefronts (recursions), each executing one outer product:

$$\mathbf{C}^{(k)} = \mathbf{C}^{(k-1)} + \mathbf{A}_k \mathbf{B}_k \tag{19}$$

or equivalently,

$$C_{ij}^{(k)} = C_{ij}^{(k-1)} + a_i^{(k)} \times b_j^{(k)} \tag{20}$$

where $a_i^{(k)} = a_{ik}$ and $b_j^{(k)} = b_{kj}$, for $k = 1, 2, \cdots, p$.

With reference to Figure 7, let us examine the computational wavefront for the first recursion in the matrix multiplication. Suppose that the internal registers of all the PEs are initially set to zero, that is, $C_{ij}^{(0)} = 0$, for all i, j. The entries of $\mathbf{A}$ are stored in the memory modules in the left (in columns), and those of $\mathbf{B}$ in the memory modules on the top (in rows). The process starts with PE(1,1), where $C_{11}^{(1)} = C_{11}^{(0)} + a_{11} \times b_{11}$. The computational activity then propagates to the neighboring PEs, (1,2) and (2,1). Since they both have input operands available (data-driven property), they will, respectively, execute $C_{12}^{(1)} = C_{12}^{(0)} + a_{11} \times b_{12}$ and $C_{21}^{(1)} = C_{21}^{(0)} + a_{21} \times b_{11}$ in parallel. The next wavefront of activity will be at PEs (3,1), (2,2), and (1,3), thus creating a computational wavefront traveling down the processor array. When the first wavefront sweeps through all the

PEs or all the available input operands of PE (1,1) are used up, the first recursion is over and the result is stored in the internal register of PE (1,1). Similarly, when the second wavefront sweeps through all the PEs, the computations of PEs (1,2) and (2,1) are completed. Next, the computations will be completed at PEs (3,1), (2,2), and (1,3). Thus, the completion of PEs will keep going and traveling down the processor array. The above concept shows that the results of the diagonal PEs (circular cells) of the processor array will be carried out at every two wavefronts after the computation of PE (1,1) is completed.

For calculating the trace computation $\mathrm{tr}(\mathbf{AB})$, we shall use the above idea to achieve this purpose. An adder tree of height $\lceil \log_2 p \rceil$, connected to the diagonal PEs of the processor array, computes the trace function in parallel with the calculation of the matrix multiplication of **AB**. This tree-structured adder architecture is also in an asynchronous data-driven manner. For example, the first adder is activated after both computations of PE (1,1) and PE (2,2) are completed. This also implies that the adders start their computations at every two wavefronts sweeping through the square processor array after the PE (1,1) is completed. The last adder will start its computation immediately after the PE $(p\,,p)$ has been completed. Thus, the result of the trace computation, $\mathrm{tr}(\mathbf{AB})$, is carried out just after the completion of the matrix operation **AB** about one addition time late. Using the above concept, we can calculate the $\mathbf{A}_s = \mathbf{A}_1\mathbf{B}_{s-1}$ and $\mathrm{tr}(\mathbf{A}_s) = \mathrm{tr}(\mathbf{A}_1\mathbf{B}_{s-1})$ of the sth iteration of Eq. (11) by letting $\mathbf{A} = \mathbf{A}_1$ and $\mathbf{B} = \mathbf{B}_{s-1}$ almost simultaneously. Note that these computations are in the RNS by using the residue arithmetic implementation. Furthermore, one may evaluate $|q_s|_{m_i} = \left||s^{-1}|_{m_i} |\mathrm{tr}(\mathbf{A}_1\mathbf{B}_{s-1})|_{m_i}\right|_{m_i}$ in the residue arithmetic modulus m_i manner by connecting a residue multipler to the (residue) adder tree, where $|s^{-1}|_{m_i}$ is a precomputed constant. The resulting $|q_s|_{m_i}$ is sent back to the diagonal PEs (circular cells) of the processor array in order to calculate the matrix diagonal addition, $|\mathbf{B}_s|_{m_i} = |\mathbf{A}_s - q_s\mathbf{I}|_{m_i}$, of the sth iteration of Eq. (11). Before performing this matrix operation, the determination of whether or not $|\mathbf{A}_s|_{m_i} = |\mathbf{A}_1\mathbf{B}_{s-1}|_{m_i}$ is a zero matrix is performed. To achieve this purpose, zero-value testing flags "*" generated by the input memory modules to test the input data of the architecture. If any PE $(i\,,j)$ has the result which is stored in the internal register R_{ij} and is not equal to zero, or either of the two inputs is the non-zero flag "Δ", then both outputs of the PE will be assigned with the non-zero flags "Δ". Otherwise, the outputs will remain to have the zero-value testing flags "*". It could be shown that if the outputs of PE $(p\,,p)$ are "*", then all the entries of $\mathbf{A}_1\mathbf{B}_{s-1}$ shall be zero, otherwise, at least one entry of $\mathbf{A}_1\mathbf{B}_{s-1}$ is non-zero. This resulting flags will be carried out immediately after the computations of all the entries of $\mathbf{A}_1\mathbf{B}_{s-1}$ are completed. In addition, the resulting flags also make the processor array and the output memory modules perform the following functions:

(1) For the processor array, if the non-zero flag "Δ" is present at each PE of the processor array, then the array performs the matrix diagonal addition, that is,

$|\mathbf{B}_s|_{m_i} = |\mathbf{A}_s - q_s\mathbf{I}|_{m_i}$. For instance, the diagonal PEs (i, i) (circular cells) execute the operation, $R_{ii} = |R_{ii} - q_s|_{m_i}$, while the off-diagonal PEs (square cells) have their same internal register values. Otherwise, all the internal registers of the PEs will be set to the initial state and the inhibited tags will be added to the output data $\mathbf{B}_s$. Once the global timing clock comes in, each column of $\mathbf{B}_s$ which is stored in the internal registers of the vertical PEs will be pumped out upward along the dotted output data line. This will make the input data and output data in the same sequencing and data format.

(2) For the output memory modules of $\mathbf{B}_{s-1}$, the entries of $\mathbf{B}_{s-2}$ coming down from the previous array are stored in the right internal buffers of the output memory modules, and the entries of $\mathbf{B}_{s-1}$ coming down from the current processor array are stored in the left internal buffers. If the resulting flag "*" is present at each memory module, then the inhibited tags will be assigned to the stored $\mathbf{B}_{s-2}$ and they indicate that those data are the desired results. Next, the data in the right internal buffers will be pumped out. Otherwise, the flag "Δ" controls the switches inside the output memory modules and forces the data stored in the left internal buffers, i.e. $\mathbf{B}_{s-1}$, to be pumped out. However, if the input $\mathbf{B}_{s-2}$ have the inhibited tags, then the above functions are disabled. The $\mathbf{B}_{s-2}$ stored in the output memory modules are in the waiting situation, before the global timing clock is present at this array.

(3) For the output memory module of q_s, its functions are very similar to the output memory modules of $\mathbf{B}_{s-1}$. Before the resulting flag comes in, the q_{s-1} from the previous array and the resulting q_s have been stored in the right and left internal buffers, respectively. If q_{s-1} has the inhibited tag, then the q_{s-1} is waited for being pumped out. Otherwise, the resulting flag "*" is present at the memory module, making the q_{s-1} being tagged. This also shows that the q_{s-1} is the desired result. Meanwhile, if the resulting flag is "Δ", then the q_s will be pumped out and the computations will continue to find the desired result. Note that the data in the above output memory modules of $\mathbf{B}_{s-1}$ and q_s are in the waiting situation, before the global timing clock is present at this array.

If we let t_a^i be the time for performing modulo m_i residue scalar addition, t_m^i be the time for performing modulo m_i residue scalar multiplication, t_d^i be the time for data transfer, and t_0^i be the time for pumping out the results. It can be shown that the residue matrix multiplication $|\mathbf{A}_1\mathbf{B}_{s-1}|_{m_i}$ can be completed in $(3p-2)(t_a^i + t_m^i + t_d^i)$ time units and the resulting residue trace computation $|\mathrm{tr}(\mathbf{A}_s)|_{m_i}$ can also be completed with $(t_a^i + t_d^i)$ time delays after the residue matrix multiplication completes its operation. Next, the residue scalar multiplication, $|q_s|_{m_i} = ||s^{-1}|_{m_i} |\mathrm{tr}(\mathbf{A}_s)|_{m_i}|_{m_i}$, and the $|q_s|_{m_i}$ can be evaluated at $[(3p-2)(t_a^i + t_m^i + t_d^i) + (t_a^i + t_d^i) + (t_m^i + t_d^i)]$ time units. One can see that the time for performing the three residue matrix functions exceeds the time for evaluating the residue matrix multiplication in a normal wavefront array by only

$(t_m^i + 2t_a^i + 3t_d^i)$ time delays. At almost the same time as the calculation of $|q_s|_{m_i}$ is completed, the resulting flags are also carried out and present at the output of PE (p, p). The resulting flags will pass through the dotted data lines, and then go back to the PEs in the array. If the resulting flag is "Δ", then a residue matrix diagonal addition, i.e., $|\mathbf{B}_s|_{m_i} = |\mathbf{A}_s - q_s\mathbf{I}|_{m_i}$, will be evaluated. Otherwise, the internal registers of the PEs will be set to the initial zero-value state. However, some modifications will be required in the first-level processor array of Figure 7 for evaluating $|\mathbf{A}_1|_{m_i}$, $|\mathbf{B}_1|_{m_i}$, and $|q_1|_{m_i}$ because $|\mathbf{A}_1|_{m_i} = |\mathbf{A}\mathbf{A}^T|_{m_i}$ is a residue non-square matrix multiplication, where $\mathbf{A}$ is a $p \times n$ matrix, and it will take $(n+2p-2)(t_a^i + t_m^i + t_d^i)$ time units to complete in a wavefront array of size $p \times p$. Using the same idea of the above discussion, the architecture can evaluate $|q_1|_{m_i}$ at time $[(n+2p-1)(t_a^i + t_m^i + t_d^i) + t_a^i + 2t_d^i]$. Meanwhile, we need both resulting data $\mathbf{A}_1$ and $\mathbf{B}_1$ so that there are two internal registers and two output data lines of each PE for having the $\mathbf{A}_1$ and $\mathbf{B}_1$ be stored in and pumped out, respectively. The other modifications for the architecture are that the $q_0 = -1$ and $\mathbf{B}_0 = \mathbf{I}$ have been stored in the right internal buffers of the output memory modules already. Otherwise, the functions of these output memory modules are similar to the output memory modules of Figure 7. Furthermore, another non-square matrix product computation $\mathbf{Y} = \mathbf{A}^T\mathbf{B}_{K-1}$ in Eq. (12) can be completed in $[(n+2p-2)(t_a + t_m + t_d)]$ time units by an ordinary fixed-point wavefront array of size $n \times p$, where t_a, t_m, and t_d are, respectively, the counterparts of t_a^i, t_m^i, and t_d^i in the ordinary fixed-point system. It should be noted that the entries of $\mathbf{Y}$ are pumped out column by column. In other words, n entries of $\mathbf{Y}$ are pumped out at the same time. So, the evaluation of the pseudo-inverse in Eq. (12), $\mathbf{A}^+ = \frac{1}{q_K}\mathbf{Y}$, may be executed in parallel by using n fixed-point division processing elements which are connected to their corresponding rows of $\mathbf{Y}$. Therefore, the parallel evaluations can be completed in $p\,(t_{div} + t_0)$ time units, where t_{div} is the time for performing a fixed-point division and t_0 is the time for pumping out the results.

In general, the performance (or throughput) of a VLSI array architecture can be evaluated by the pipelined time, denoted by T, which is the time interval between two successive computations for the architecture. In our case, the pipelined time of the proposed two-level macro-pipelined array is equal to the global timing clock period. Because of achieving the synchronization of the pipe, the clock period is always equal to the largest processing time of the first-level processor arrays plus the time for pumping out the results. Since the Decell algorithm is implemented in the L-modulus arithmetic system, so L two-level macro-pipes are required for the realization of the L identical parallel Decell algorithms with different moduli m_i, $i = 1, \cdots, L$. Thus, the maximum clock period of the ith parallel macro-pipe, $1 \leq i \leq L$, will synchronize the multiple-pipe structure

$$T = \max\{ \max_{1 \le i \le L} \{ [(3p-1)(t_a^i + t_m^i + t_d^i) + t_a^i + 2t_d^i + t_0^i], \\ [(n+2p-1)(t_a^i + t_m^i + t_d^i) + t_a^i + 2t_d^i + t_0^i] \}, \\ [(n+2p-2)(t_a + t_m + t_d)], p[(t_{div} + t_0)] \}$$

As an example, for a 12-link redundant robot, the associated Jacobian is a 6×12 matrix, and its pipelined time is

$$T = \max\{ \max_{1 \le i \le L} (24t_a^i + 23t_m^i + 25t_d^i + t_0^i), 22(t_a + t_m + t_d), p(t_{div} + t_0) \}$$

Next, we shall give some numerical data on evaluating the performance of the proposed architecture for computing the pseudo-inverse Jacobian for a 12-link redundant robot. If the input data size to the proposed architecture is 16 bits, then from Table 1, the suggested number of parallel pipes, L, is equal to 4, the largest modulus bit size, g, is equal to 5, and the dynamic range M becomes 392863. Though it is not required to use moduli of the same bit length, the choice of moduli rests on the desired dynamic range M and the number of parallel pipes L. For example, from Table 2, a four moduli $\{31, 29, 23, 19\}$ is chosen to be greater than $p(=6)$ which yields a dynamic range of 392863 when a 16-bit input number is used. Chiang and Johnsson [23] estimated the computational speeds of their residue adder and residue multiplier designs (in 4 μm NMOS) and showed that the k-bit residue adder and the k-bit residue multiplier, respectively, require $10k \sim 15k$ ns and $30k \sim 45k$ ns. Using these data, the upper computational time bounds for a 5-bit residue adder and a 5-bit residue multiplier are, respectively, equal to $75ns$ and $225ns$. On the other hand, if we were to use a 16-bit fixed-point adder and multiplier implemented in a 1.5 μm CMOS integrated circuit, a 46.8 ns addition time and a 64 ns multiplication time were reported [24]. For simplicity, if we do not consider the time for data transfer and I/O interface and if we let $t_{div} = t_m = 64$ ns, the pipelined time for the proposed array architecture becomes $T = \max(6975, 2437.6, 384)$ $ns = 6.975$ μs. Another desired performance measure to evaluate the proposed array architecture is the initial delay time (or set-up time) which is defined as the required time interval for obtaining the first resulting pseudo-inverse Jacobian from the architecture. There are $(p+4)$ processing stages in the proposed array architecture: p stages for evaluating Eq. (11), two stages for evaluating Eq. (12), and two stages for dealing with arithmetic decomposition and recomposition. Hence, the initial delay time is equal to $(p+4) \times T = 10 \times 6.975$ $\mu s = 69.75$ μs, when $p = 6$. The above performance evaluations indicate that the real-time computation of a general pseudo-inverse Jacobian matrix is achievable with current VLSI custom or semi-custom design technology.

7. Conclusion

A two-level macro-pipelined VLSI array architecture has been designed, based on the iterative Decell algorithm, for the real-time computation of the *exact* solution of a general $p \times n$ ($n \geq p$) pseudo-inverse Jacobian matrix. Due to the ill-conditioned problem in determining the rank of the manipulator Jacobian, the residue arithmetic is employed to obviate the round-off error occurred in the Decell algorithm computations in order to obtain an *exact* solution. This is achieved by using the residue arithmetic in determining whether or not the matrix product $\mathbf{A}_1 \mathbf{B}_K$ is *exactly* a zero matrix in the $(K+1)$th iteration of the Decell algorithm. The first-level arrays of the proposed architecture are asynchronous data-driven, wavefront-like two-dimensional arrays which perform the matrix multiplications, matrix diagonal additions, and trace computations in parallel. A pool of the first-level arrays (including the arithmetic decomposition and recomposition cells) are then configured into a second-level macro-pipeline of $(p+4)$ stages with outputs of one array acting as inputs to another array in the pipe. The second-level linear array is a synchronous systolic array whose data movements in the array are controlled by global timing-reference clock pulses. The pipelined time of the proposed two-level pipelined array architecture has a computational order of $O(n+2p-1)$ which is the same computational complexity order as evaluating a matrix product in an ordinary wavefront array. For a 12-link redundant robot, a pipelined time of 6.975 μs is achievable with current VLSI custom design technology.

8. References

[1] D. E Whitney, "Resolved Motion Rate Control of Manipulators and Human Protheses," *IEEE Transactions on Man-Machine Systems,* Vol. MMS-10, No. 2, pp. 47-53, June 1969.

[2] J. Y. S. Luh, M. W. Walker, and R. P. Paul, "Resolved-Acceleration Control of Mechanical Manipulators," *IEEE Trans. on Automatic Control,* Vol. AC-25, No. 3, pp. 468-474, Nov. 1980.

[3] C. S. G. Lee and B. H. Lee, "Resolved Motion Adaptive Control for Mechanical Manipulators," *Trans. of ASME, J. Dynamic Syst., Meas., Contr.,* Vol. 106, No. 2, pp. 134-142, June 1984.

[4] R. P. Paul, B. E. Shimano, and G. Mayer, "Differential Kinematic Control Equations for Simple Manipulators," *IEEE Transactions of Systems, Man, and Cybernetics,* Vol. SMC-11, no. 6, 1981.

[5] R. Featherstone, "Position and Velocity Transformations between Robot End-Effector Coordinates and Joint Angles," *Int. J. Robotics Research,* Vol. 2, No. 2, Summer 1983.

[6] M. B. Leahy et. al, "Efficient PUMA Manipulator Jacobian Calculation and Inversion," *J. Robotic Systems,* Vol. 4, No. 2, 1987.

[7] D. E. Orin and W. W. Schrader, "Efficient Computation of the Jacobian for Robot Manipulator," *Int'l J. Robotics Research,* Vol. 3, No. 4, Winter 1984.

[8] D. E. Orin et al, "Systolic Architectures for Computation of the Jacobian for Robot Manipulator," *Computer Architectures for Robotics and Automation,* Gordon and Breach, NY, 1987.

[9] T. B. Yeung and C. S. G. Lee, "Efficient Parallel Algorithms and VLSI Architectures for Manipulator Jacobian Computation," Technical Report TR-EE 87-16, School of Electrical Engineering, Purdue University, May 1987.

[10] C. A. Klein and C. H. Huang, "Review of Pseudo-inverse Control for Use with Kinematically Redundant Manipulators," *IEEE Trans. on System, Man and Cybernetics,* Vol. SMC-13, No. 13, pp. 245-280, March/April 1982.

[11] K. C. Suh and J. M. Hollerbach, "Local versus Global Torque Optimization of Redundant Manipulators," *IEEE J. Robotics and Automation,* Vol. RA-3, No. 4, Aug. 1987.

[12] C. R. Rao and S. K. Mitra, *Generalized Inverse: Theory and Applications,* Wiley, NY, 1974.

[13] S. L. Campbell and C. D. Meyer, Jr., *Generalized Inverse of Linear Transformations,* pp. 242-248, Pitman, London, 1979.

[14] W. T. Stallings and T. L. Boullion, "Computation of Pseudo-Inverse Matrices Using Residue Arithmetic," *SIAM Review,* Vol. 14, pp. 152-163, 1972.

[15] T. M. Rao et al., "Residue Arithmetic Algorithms for Exact Computation of generalized-inverse of Matrices," *SIAM J. Numer. Anal.,* Vol. 13, pp. 155-171, 1976.

[16] R. T. Gregory and E. V. Krishnamurthy, *Methods and Applications of Error-Free Computation,* Springer-Verlag, NY, 1984.

[17] S. Y. Kung et al., "Wavefront Array Processor: Language, Architecture, and Applications," *IEEE Trans. on Computer,* Vol. C-31, No. 11, pp. 1054-1066, Nov. 1982.

[18] B. W. Lamacchia and G. R. Redinbo, "RNS Digital Filtering Structures for Wafer-Scale Integration," *IEEE J. on Selected Area in Communications,* Vol. SAC-4, No. 1, Jan. 1986.

[19] M. A. Soderstrand, *Residue Number System Arithmetic: Modern Applications in Digital Signal Processing,* The IEEE Inc., 1986.

[20] N S. Szabo and R. I. Tanaka, *Residue Arithmetic and its Applications to Computer Technology,* McGraw-Hill, NY, 1967.

[21] G. A. Jullien, "Implementation of Multiplication Modulo A Prime Number, with Applications to Number Theoretic Transforms," *IEEE Trans. Comput.,* Vol. C-29, No. 10, pp. 899-905, Oct. 1980.

[22] F. J. Mowle, *A Systematic Approach to Digital Logic Design,* Reading, Addison-Wesley, MA, 1983.

[23] C. L. Chiang and L. Johnsson, "Residue Arithmetic and VLSI," *Proc. of 1983 Int'l Conf. Comput. Design,* pp. 80-83, 1983.

[24] D. A. Henlin et al., "A 16 bit $\times$ 16 bit Pipelined Multiplier Macrocell," *IEEE J. of Solid-State Circuits,* Vol. SC-20, No. 2, pp. 542-547, April 1985.

Table 1 L, g, and M for various input bit size

n	L	g	M
8	2	5	899
16	4	5	392863
32	5	7	17.2×10^{9}
64	10	7	9.98×10^{19}

n is the input bit size.
L is the number of parallel pipes.
g is the largest modulus bit size.
M is the dynamic range.

Table 2. Prime Numbers from 1 to 128

Bits	Primes
3	2, 3, 5, 7
4	11, 13
5	17, 19, 23, 29, 31
6	37, 41, 43, 47, 53, 59, 61
7	67, 71, 73, 79, 83, 89, 97, 101, 103, 107, 109, 113, 127

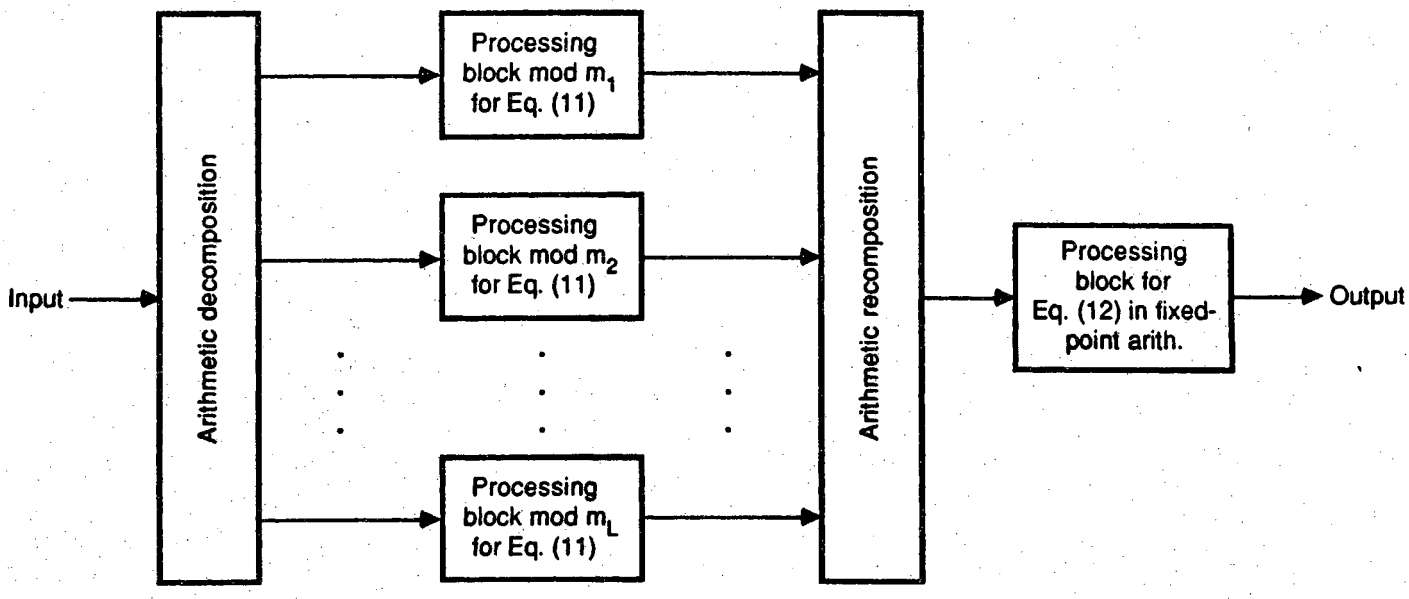


Figure 1. General residue arithmetic array architecture for the Decell algorithm.

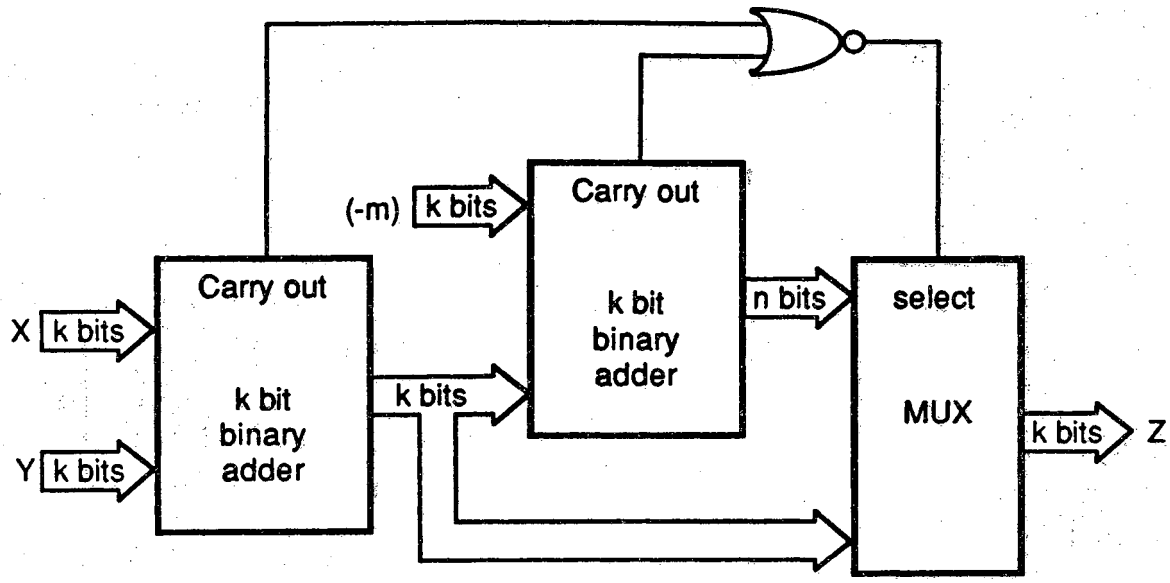


Figure 2. Implementation of residue adder with two k-bit binary adders.

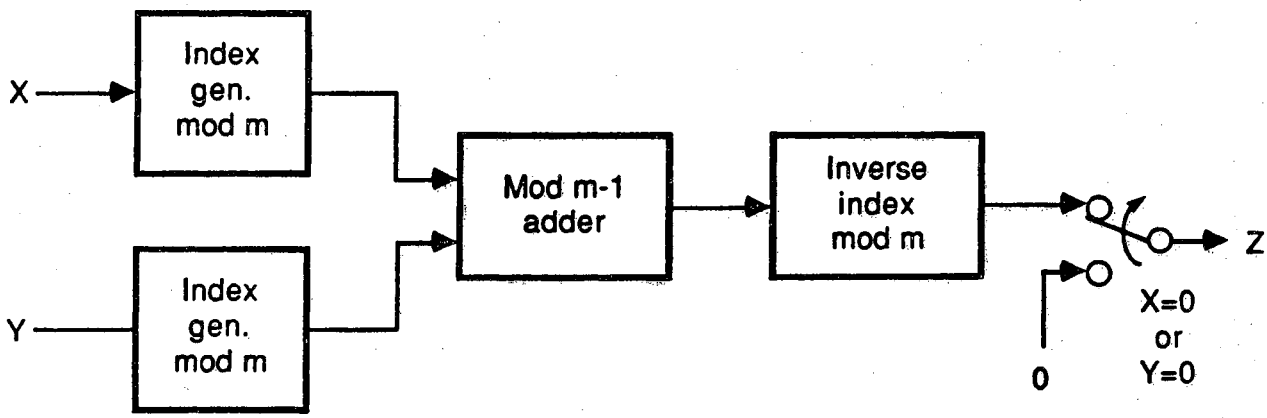


Figure 3. Implementation of residue multiplier using index addition.

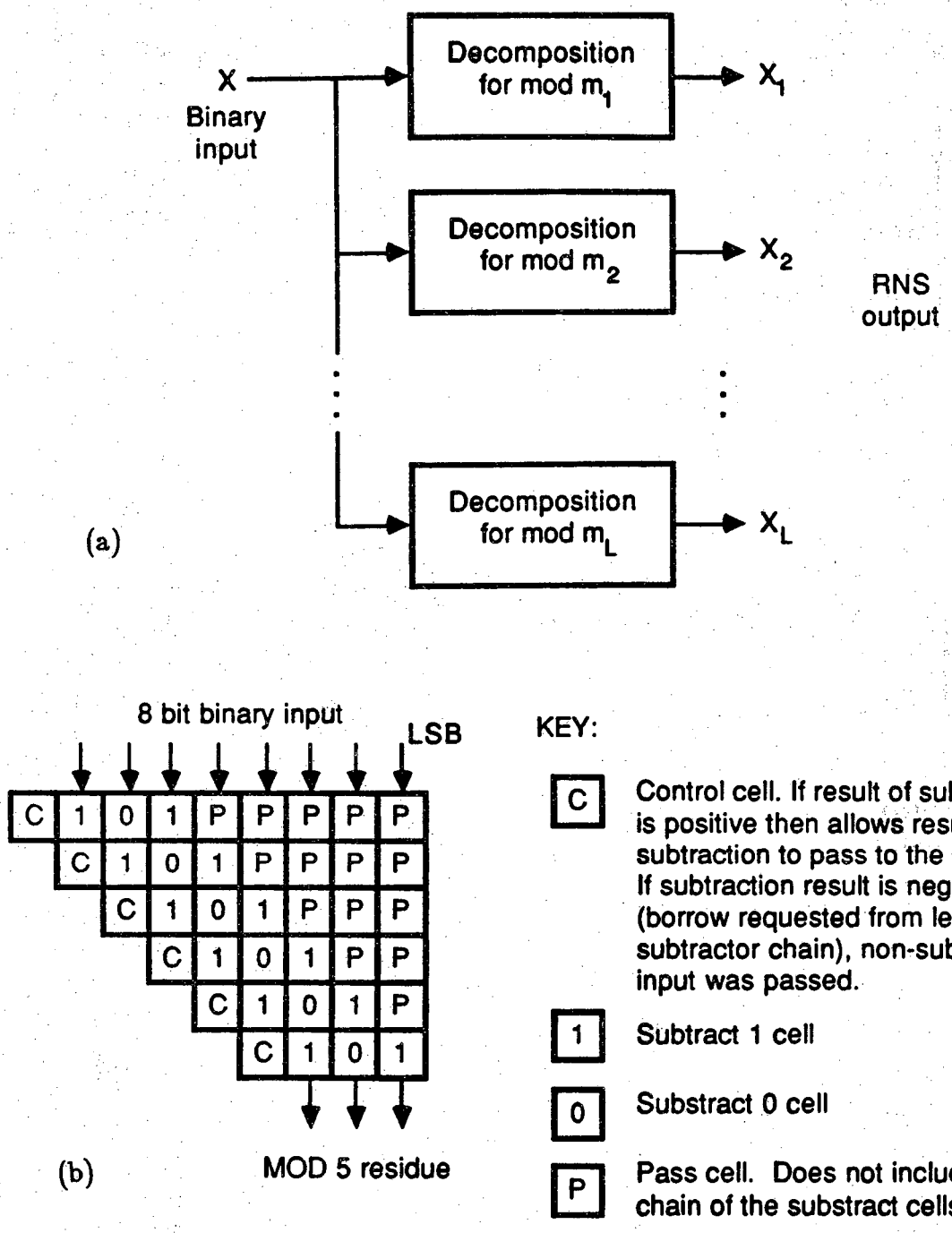


Figure 4. (a) General decomposition architecture.
(b) Residue decomposer (modulo 5) floor plan.

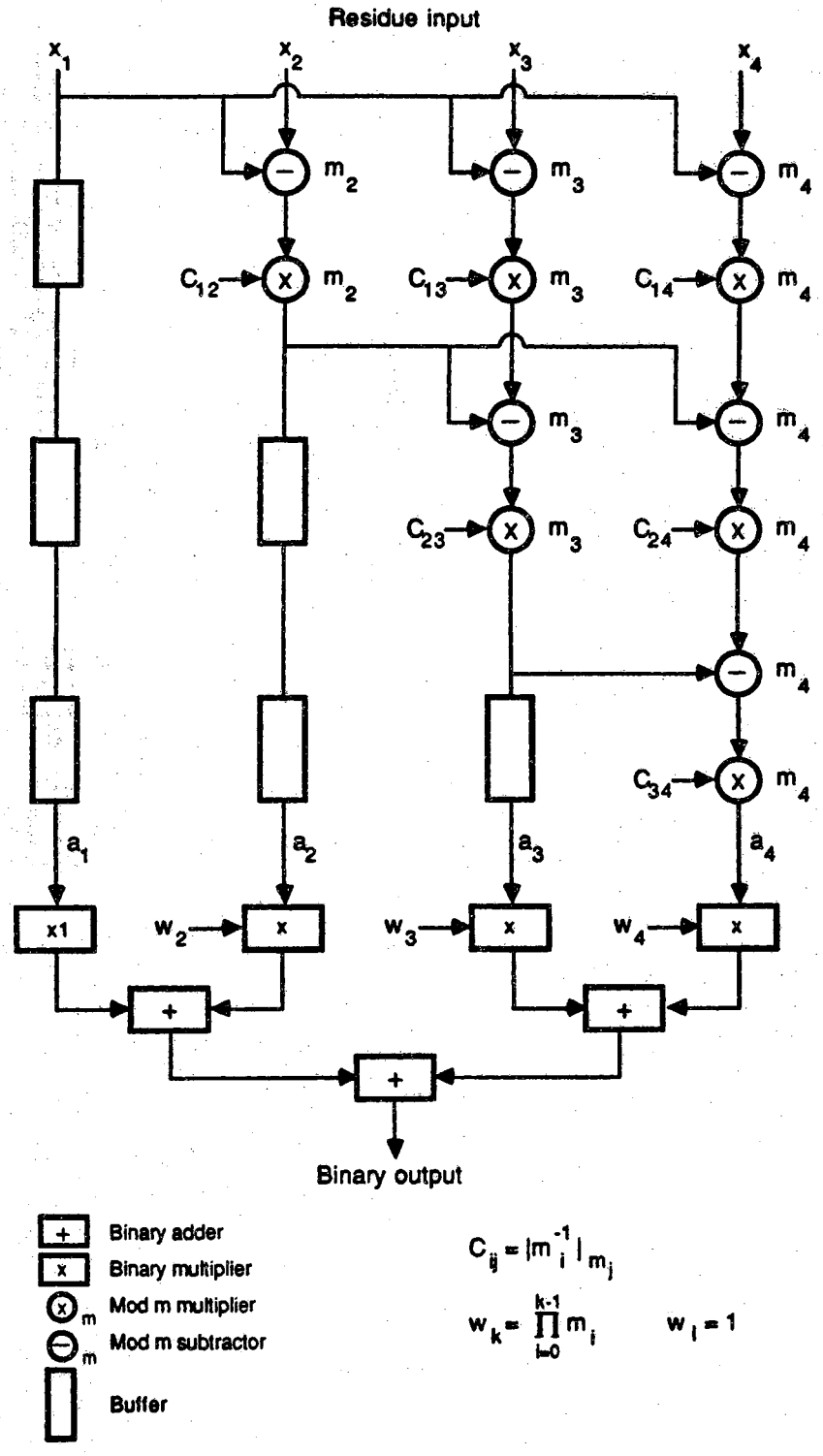
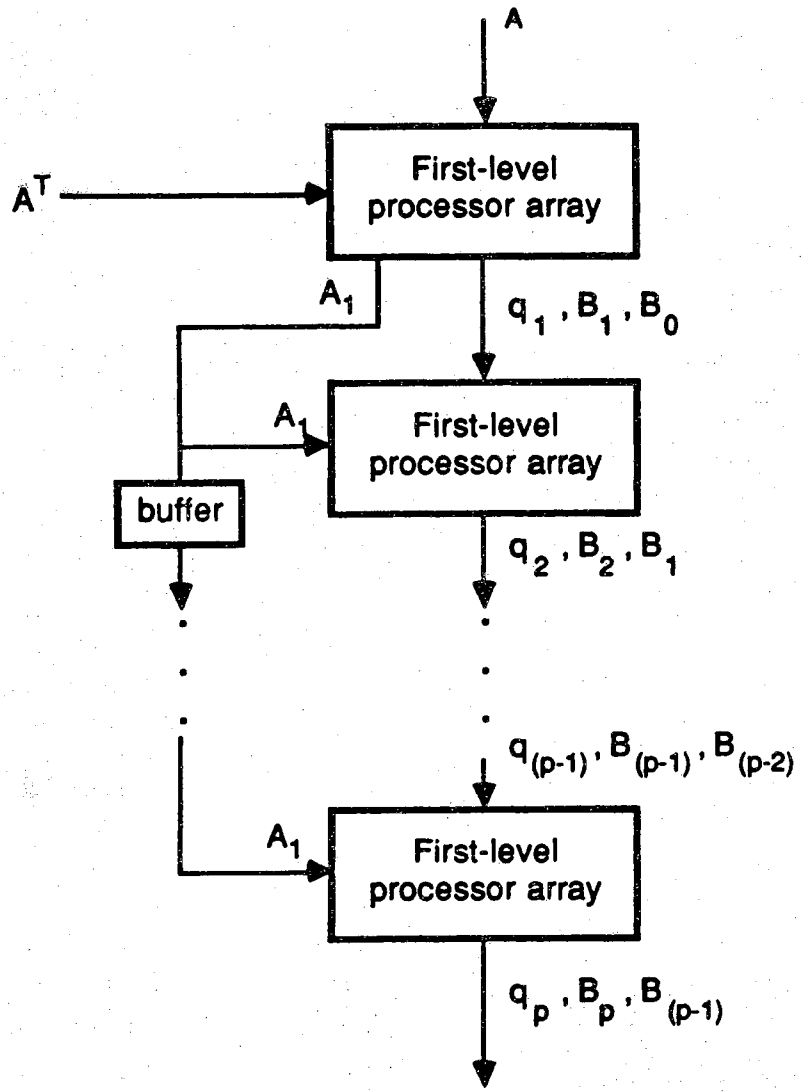


Figure 5. Mixed-radix recomposition pipelined architecture.



Note that: the first-level processor array will be implemented in Fig. 7.

Figure 6. The second-level p-stage linear pipelined array for equation (11).

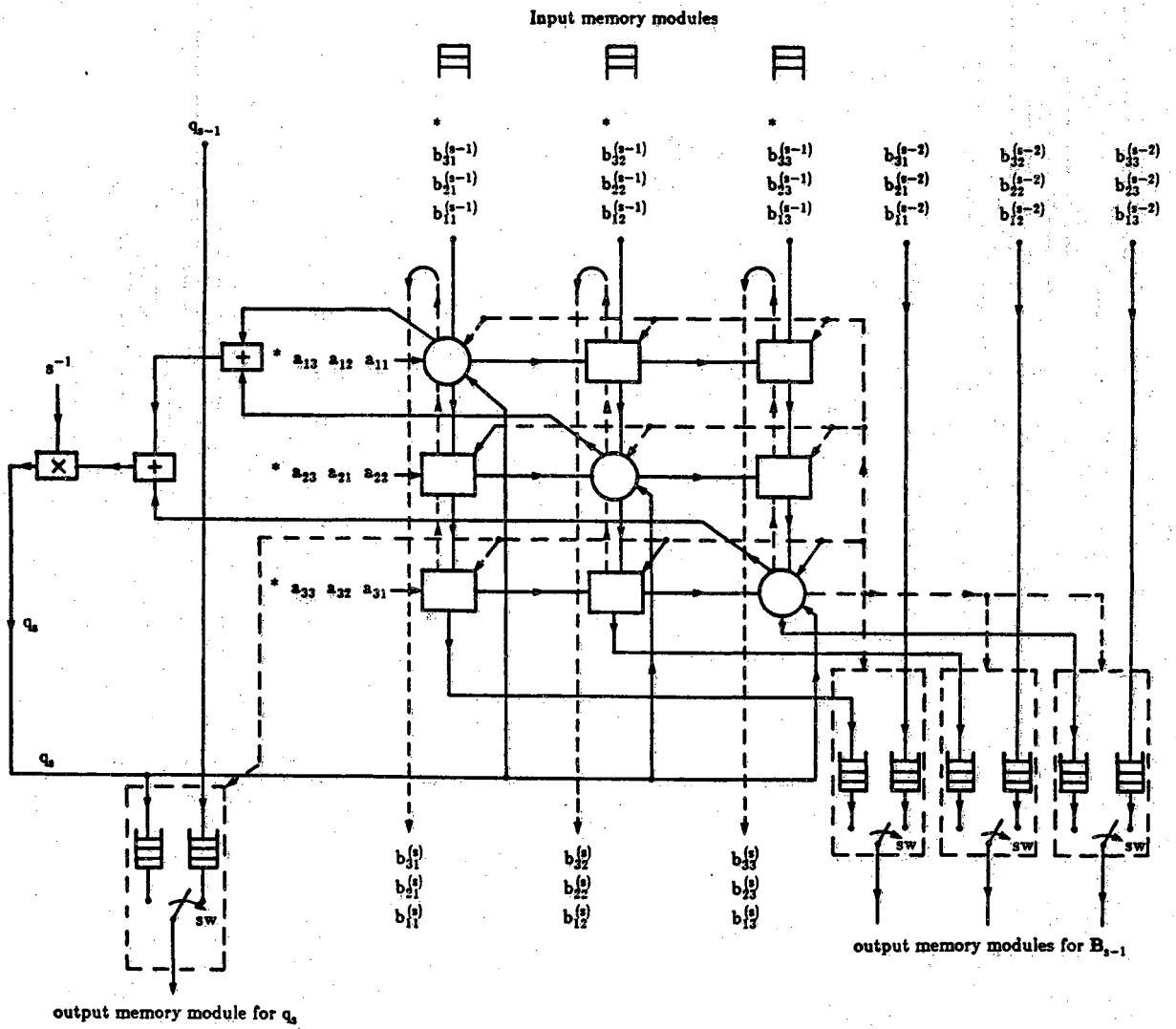
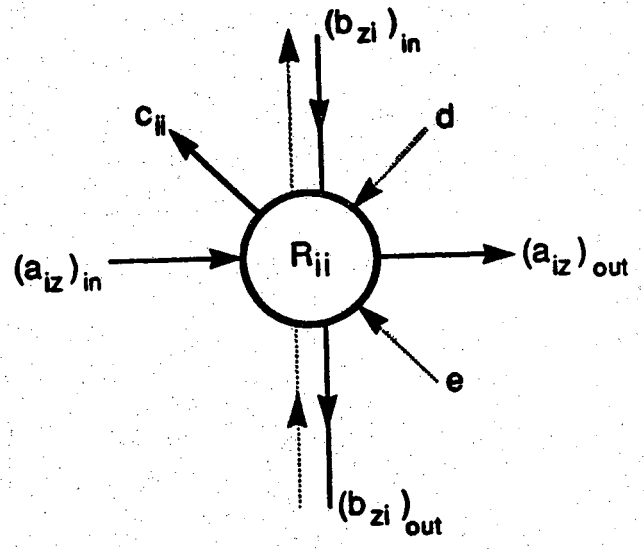


Figure 7. (a) An asynchronous data-driven wavefront-like processor array for the sth iteration of equation (11).



A. (1) If input data $(b_{zi})_{in}$ have inhibited tags, then the PE (i,i) is disabled.
(2) Initial: Internal register $R_{ii} = 0$.

B. The PE $(i\,,i)$ is activated, when the following conditions are satisfied.
(1) Both $(a_{iz})_{in}$ and $(b_{zi})_{in}$ are available at the PE $(i\,,i)$, thus performing
(i) $R_{ii} \leftarrow R_{ii}$, if either of $(a_{iz})_{in}$, $(b_{zi})_{in} = *$ *or* Δ *or null*
(ii) Otherwise $R_{ii} \leftarrow R_{ii} + (a_{iz})_{in} \times (b_{zi})_{in}$
or
(2) Both d and e are available at PE (i,i), thus performing

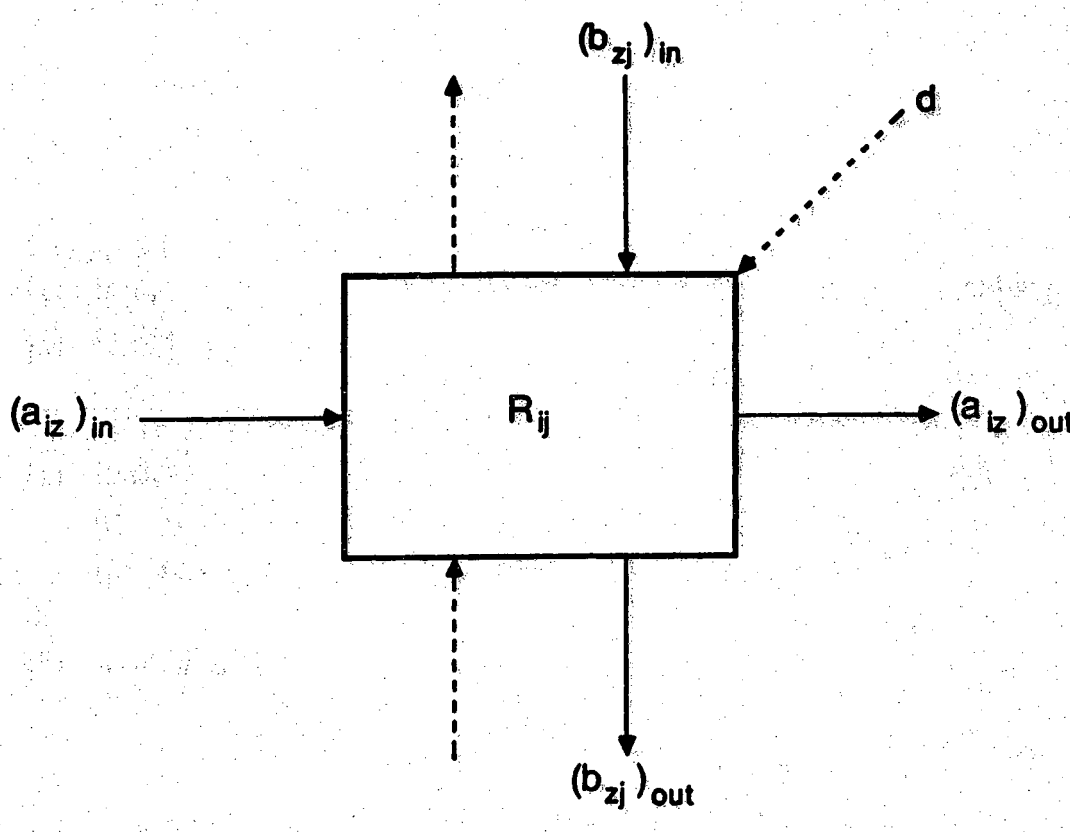
$$R_{ii} \leftarrow \begin{cases} R_{ii} - e\,, & \text{if } d = \Delta \\ 0\,, & \text{if } d = * \end{cases}$$

C. After PE (i,i) has been executed p times, the output c_{ii} will be activated and equal to the accumulated value in the internal register $R_{ii} (= \sum_{z=1}^{p} a_{iz} \times b_{zi})$.

D. I/O data token operations:
(1) $(a_{iz})_{out} = \Delta$ and $(b_{zi})_{out} = \Delta$, if $R_{ii} \neq 0$ or $(a_{iz})_{in} = \Delta$ or $(b_{zi})_{in} = \Delta$.
(2) Otherwise $(a_{iz})_{out} = (a_{iz})_{in}$, and $(b_{zi})_{out} = (b_{zi})_{in}$.
Notice that "$*$" is a zero value testing flag and "Δ" is a non-zero flag.

E. Once the global timing clock comes in, the data stored in the current and previous PEs will be pumped out along the dotted output data line.

Figure 7. (b) The diagonal PE (i,i) (circular cells).



A. (1) If input data $(b_{zj})_{in}$ have inhibited tags, then the PE (i,j) is disabled.

(2) Initial: Internal register $R_{ij} = 0$.

B. The PE (i,j) is activated, when the following conditions are satisfied.

(1) Both $(a_{iz})_{in}$ and $(b_{zj})_{in}$ are available at PE (i,j), thus performing

(i) $R_{ij} \leftarrow R_{ij}$, if either of $(a_{iz})_{in}$ and $(b_{zj})_{in} = *$ or Δ or null.

(ii) Otherwise $R_{ij} \leftarrow R_{ij} + (a_{iz})_{in} \times (b_{zj})_{in}$

or

(2) when d is present at PE (i,j), thus performing

$$R_{ij} \leftarrow \begin{cases} R_{ij}, & \text{if } d = \Delta \\ 0, & \text{if } d = * \end{cases}$$

C. I/O data token operations:

(1) $(a_{iz})_{out} = \Delta$ and $(b_{zj})_{out} = \Delta$, if $R_{ij} \neq 0$ or $(a_{iz})_{in} = \Delta$ or $(b_{zj})_{in} = \Delta$

(2) Otherwise $(a_{iz})_{out} = (a_{iz})_{in}$, and $(b_{zj})_{out} = (b_{zj})_{in}$.

D. Once the global timing clock comes in, the data stored in the current and previous PEs will be pumped out along the dotted output data line.

Figure 7. (c) The off-diagonal PE (i,j) (square cells).